Caixa de ferramentas da Arquitetura de Software

Como tornar suas aplicações mais escaláveis, confiáveis e seguras

Casa do Código | alura

EDUARDO FELIPE ZAMBOM SANTANA

Sumário

- ISBN
- Agradecimentos
- Sobre o livro
- Sobre o autor
- Prefácio
- Parte 1: Introdução à Arquitetura de Software
  - 1 Arquitetura de software
  - 2 Modelos de Arquitetura de Software
- Parte 2: Desempenho
  - 3 Desempenho
  - 4 Desempenho e escalabilidade no código
  - 5 Desempenho e escalabilidade na infraestrutura
- Parte 3: Confiabilidade
  - 6 Confiabilidade
  - 7 Confiabilidade no código
  - 8 Confiabilidade na infraestrutura
- Parte 4: Segurança
  - 9 Segurança
  - 10 Segurança no código
  - 11 Segurança na infraestrutura
- Parte 5: Outros requisitos não funcionais
  - 12 Outros requisitos

ISBN

Impresso: 978-85-5519-367-5

Digital: 978-85-5519-365-1

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Santana, Eduardo Felipe Zambom
Caixa de ferramentas da arquitetura de software : como tornar suas aplicações mais escaláveis, confiáveis e seguras / Eduardo Felipe Zambom Santana. -- São Paulo : AOVS Sistemas de Informática, 2024.

ISBN 978-85-5519-367-5

1. Arquitetura de software 2. Arquitetura (Tecnologia do Ambiente) 3. Software de aplicação - Desenvolvimento I. Título.

24-193558 CDD-004.2

Índices para catálogo sistemático:

1. Software : Arquitetura, projeto e desenvolvimento : Especificações técnicas : Ciência da computação 004.2

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Agradeço à minha família, meus pais, Luiz e Sonia, e meus irmãos, Luiz e Bruna. Sem eles, dificilmente teria chegado até aqui.

Agradeço à minha namorada, Brianda, que me acompanha em praticamente toda a minha vida profissional.

Agradeço também ao pessoal da Casa do Código, especialmente ao Antonio, pela ajuda e revisão na escrita deste livro.

Sobre o livro

Em um projeto de software, as pessoas desenvolvedoras se preocupam principalmente com o código, que vai executar algum requisito pedido pelos clientes da aplicação. Porém, se a aplicação começa a crescer, tendo que lidar com um grande número de usuários e com uma grande quantidade de dados para processar, não será suficiente apenas desenvolver o código e esperar que ele vá continuar funcionando sempre da mesma maneira, pois diversos problemas, como lentidão e indisponibilidade, podem começar a ocorrer.

Para lidar com o crescimento da aplicação, teremos que pensar em diversas estruturas para garantir que ela continue funcionando mesmo em condições muito mais complexas. Para isso, temos que pensar na **Arquitetura de Software**, que define a estrutura geral de um sistema para que ele consiga atender diversos requisitos que são comuns à grande maioria das aplicações, que são os requisitos não funcionais (ou atributos de qualidade).

Os requisitos não funcionais definem uma série de atributos que as aplicações devem atender para que ela tenha a qualidade esperada pelos seus usuários. Existe um grande número de requisitos não funcionais. Alguns dos mais conhecidos são o desempenho, a confiabilidade e a segurança. Uma aplicação que tenha um bom desempenho deve ser também escalável, isto é, continuar funcionando mesmo que o número de transações cresça. Uma aplicação também deve ser tolerante a falhas, não permitindo que uma falha cause uma indisponibilidade do serviço.

É praticamente impossível atingir todos os requisitos não funcionais de que uma aplicação necessita pensando apenas em código. Por isso, é essencial que desenvolvedores e desenvolvedoras saibam identificar implementações que podem causar problemas, seja de desempenho, de confiabilidade ou de segurança. Além disso, é essencial conhecer um grande número de ferramentas e tecnologias

que podem ser usadas para evitar problemas, ou até para resolver um problema que talvez já esteja ocorrendo em uma aplicação.

Este livro tem o objetivo de apresentar e discutir a pessoas desenvolvedoras, sejam iniciantes ou já seniores, os principais requisitos não funcionais que devem ser pensados na hora de implementar uma aplicação. Entre esses requisitos estão o desempenho, a confiabilidade e a segurança. Dentro desses três requisitos, serão discutidos diversos outros que são derivados ou bastante relacionados a eles, como escalabilidade, disponibilidade, elasticidade, monitoramento, controle de acesso e tolerância a falhas. Além de discutir os requisitos não funcionais, também serão apresentadas diversas ferramentas para a implementação desses requisitos, como *cache*, balanceadores de carga e ferramentas de integração contínua.

Como este livro está organizado?

O livro tem dois objetivos principais: apresentar as arquiteturas de software mais utilizadas e discutir os principais requisitos não funcionais que devem ser considerados na maioria das aplicações. Para isso, este livro está organizado em cinco partes.

A Parte 1, *Introdução à Arquitetura de Software*, descreve algumas das arquiteturas de software mais utilizadas. Elas se dividem em dois tipos principais: os *monólitos* e as *arquiteturas distribuídas*. Dentro desses dois tipos existem diversas variações, como monólitos que são implementados usando a arquitetura em camadas, ou arquiteturas distribuídas que usam eventos ou microsserviços. Além de apresentar os tipos de arquitetura, também serão discutidas as vantagens e desvantagens de cada um dos modelos em relação aos requisitos não funcionais.

A Parte 2, *Desempenho*, apresentará formas para melhorar o desempenho da aplicação. Veremos diversos padrões e tecnologias que podem ajudar uma aplicação a alcançar um bom desempenho, como a utilização de *caches* e de algoritmos que possuem baixa complexidade de execução. Além disso, serão discutidos outros dois requisitos não funcionais que são bastante ligados ao desempenho: a *escalabilidade* e a *elasticidade*.

A Parte 3, *Confiabilidade*, mostrará ferramentas para deixar a aplicação mais confiável e evitar, ou pelo menos minimizar, os riscos de os usuários serem impactados com eventuais falhas do sistema. Também serão mostradas ferramentas para o monitoramento da aplicação, para que, quando uma falha acontecer, ela seja detectada o mais rápido possível.

A Parte 4, *Segurança*, focará na segurança das aplicações. Existem diversos padrões que devem ser seguidos, desde como os servidores devem ser expostos na internet, até como os dados dos usuários devem ser mantidos em bancos de dados para evitar riscos em uma aplicação. Serão apresentados alguns dos conceitos mais importantes para a segurança de aplicações, como o controle de acesso e o monitoramento de dependências.

Por fim, a Parte 5, *Outros requisitos não funcionais*, discutirá requisitos não funcionais como a usabilidade e a testabilidade, entre outros que também são bastante importantes, mas que não serão abordados com muita profundidade nas Partes 2, 3 e 4 por não serem tão ligados à arquitetura e à execução das aplicações.

Sobre o autor

Tenho quase 20 anos de experiência com Ciência da Computação, especialmente nas áreas de Engenharia de Software e Sistemas Distribuídos. Trabalhei principalmente com Java, tendo pego praticamente toda a evolução da plataforma, começando no Java 1.2 e agora já usando a versão 21. Também trabalhei com praticamente todos os principais frameworks dessa linguagem, como os já legados Struts e JSF e, atualmente, com Spring e Quarkus. Além do Java, já fiz bastante coisa em Python, JavaScript, Golang e Erlang.

Já trabalhei em diversas empresas, como o eBay e o Serasa; atualmente, sou Arquiteto de Software na BotCity. Fiz graduação e mestrado, em 2007 e 2010, respectivamente, na Universidade Federal de São Carlos (UFSCar) e doutorado na Universidade de São Paulo (USP) em 2019, trabalhando principalmente com Sistemas Distribuídos.

Se você quiser entrar em contato comigo, fique à vontade para me chamar no LinkedIn ou no GitHub:

- https://www.linkedin.com/in/ezambomsantana/
- https://github.com/ezambomsantana

Prefácio

Este livro é resultado dos quase 20 anos de trabalho com Engenharia de Software, alguns como desenvolvedor, outros como arquiteto, alguns também como professor, e todos como aluno desta área, tão importante na Ciência da Computação. Muitas das coisas que eu coloquei neste livro eu gostaria de ter aprendido há 15 anos, para utilizar em alguns projetos que participei. Com a experiência que tenho hoje, fica clara a falta de preocupação com diversas partes essenciais de uma aplicação, como com a sua confiabilidade e sua segurança.

Já trabalhei em projetos que eram excelentes produtos, que foram validados com os clientes e tinham tudo para ser um grande sucesso, mas que falharam, seja porque atingiram um número maior de clientes e não suportaram a carga, ou porque não eram confiáveis o suficiente. Alguns até funcionavam para os casos de uso básico para os quais foram projetados, mas qualquer coisa que fugia um pouco do padrão fazia a aplicação apresentar resultados inesperados.

Sem dúvida, os assuntos que este livro foca — desempenho, confiabilidade e segurança — são muito mais discutidos hoje do que eram quando eu comecei a trabalhar na área, e a quantidade de ferramentas e técnicas que existem hoje para conseguir implementar esses requisitos com sucesso é muito maior. Quem me dera que, há 10 anos, quando trabalhei em um projeto que tinha que atender uma fila de tickets de atendimento, já existisse o Apache Kafka.

Apesar da grande evolução que essa área teve nos últimos anos, a implementação desses requisitos é tão ou até mais desafiadora do que era quando eu comecei. Isso porque as aplicações hoje são maiores e mais complexas, possuem um número de usuários que pode ser muito maior e estão muito mais expostas na internet, o que faz os desafios de segurança serem muito maiores.

Espero que você aproveite a leitura e que os conhecimentos contidos neste livro sejam úteis para que a sua aplicação funcione com um bom desempenho e de forma confiável e segura.

— *Eduardo Zambom*

Parte 1: Introdução à Arquitetura de Software

Capítulo 1
Arquitetura de software

Existem diversos tipos de sistemas, desde aplicações muito simples, que possuem apenas cadastros simples, com alguns formulários e listagens de dados, até aplicações bastante complexas, que precisam processar milhares ou milhões de requisições em um curto período de tempo, fazendo integrações com outros sistemas e sincronizando diversas transações que acontecem paralelamente.

Em aplicações mais simples, pensar na *arquitetura do software* talvez não seja muito importante. Para uma simples aplicação web, talvez executá-la em um *servidor na nuvem*, desenvolvê-la em um *monólito* e implementá-la com acesso a um *banco de dados relacional* seja mais do que o suficiente para a execução com uma disponibilidade em praticamente 100% do tempo.

Em aplicações mais complexas, essa arquitetura simples descrita no parágrafo anterior pode até funcionar durante algum tempo, mas, com o crescimento do número de usuários ou com a inclusão de funcionalidades que demandem grande poder computacional, a aplicação pode começar a falhar. Isso pode acontecer porque os recursos computacionais usados não são suficientes para satisfazer a demanda que a aplicação necessita.

A primeira solução para resolver esse problema é simplesmente melhorar os recursos computacionais para a aplicação: aumentar a capacidade de processamento e a quantidade de memória disponível. Isso pode funcionar por algum tempo, mas logo os custos ficarão altos demais. Por isso, em aplicações mais complexas, é necessário projetar e implementar uma arquitetura mais *robusta*,

para que a aplicação continue a ter um bom desempenho, uma alta disponibilidade e uma boa segurança, mesmo em condições adversas, e sem ter um custo extremamente alto, tanto para o desenvolvimento quanto para a sua execução.

A decisão de como fazer essas implementações necessárias para melhorar a execução das aplicações faz parte do que chamamos de **arquitetura de software**. Descrever exatamente o que é arquitetura de software é uma tarefa difícil. Até hoje não existe uma definição universalmente aceita para esse conceito. Duas bastante conhecidas são (ambas em tradução livre do original em inglês):

- "***Arquitetura de software é a estrutura fundamental de um sistema, incorporada em seus componentes, suas relações entre si e com o ambiente, e os princípios que governam seu projeto e evolução***" (do documento *1471-2000 - IEEE Recommended Practice for Architectural Description for Software-Intensive Systems*; IEEE, 2000);
- "***Arquitetura de software é o conjunto de estruturas necessárias para se pensar sobre o sistema, que compreende elementos de software, as relações entre eles e as propriedades de ambos***" (do livro *Documenting Software Architectures: Views and Beyond*; CLEMENTS *et al.*, 2010).

Note que essas definições focam bastante na descrição dos componentes da aplicação e de como eles se comunicam. A essas definições, eu adiciono que, além de definir os componentes e suas formas de comunicação, a arquitetura de software deve definir como a aplicação vai lidar com os principais *requisitos não funcionais* que ela precisa atender.

Na Engenharia de Software, os **requisitos funcionais** são as funcionalidades que a aplicação precisa implementar. Por exemplo, o cadastro e listagem de usuários e a realização de compras são

requisitos que precisam ser implementados para que uma aplicação atenda ao que um usuário espera.

Porém, não adianta nada uma aplicação atender a todos os requisitos que o usuário deseja, mas o funcionamento dela não ser bom — o sistema demorar muito para responder uma chamada, ficar fora do ar por muito tempo ou ter uma interface muito complexa, que dificulta a utilização da aplicação. Esses *atributos de qualidade*, que são independentes da *funcionalidade* de um sistema, são chamados de **requisitos não funcionais**.

Alguns dos requisitos não funcionais mais importantes são:

- **Desempenho:** a aplicação deve responder em um tempo que seja aceitável para o usuário. Por exemplo, uma página web não pode demorar mais do que alguns poucos segundos para carregar, senão o usuário ficará esperando por muito tempo;
- **Escalabilidade:** normalmente, uma aplicação sempre funcionará bem se o número de usuários for pequeno, mas, dependendo do tipo de aplicação, ela deve estar preparada também para o aumento repentino da carga. Uma aplicação é *escalável* quando, mesmo com o aumento no número de usuários (ou transações), ela suportar a demanda sem que o seu desempenho se deteriore;
- **Elasticidade:** alguns tipos de aplicações têm uma variabilidade muito grande no número de usuários em diferentes períodos. Por exemplo, uma aplicação que gera rotas no mapa de uma cidade terá muito mais requisições nos horários de picos de trânsito. Nesses casos, os recursos computacionais usados devem flutuar de acordo com a demanda, para evitar que a aplicação desperdice recursos em períodos de pouco uso ou tenha problemas de desempenho ou escalabilidade por falta de recursos nos horários de pico;
- **Confiabilidade:** uma aplicação confiável deve sempre retornar uma resposta válida quando for requisitada. Uma

disponibilidade de 100% é praticamente impossível, mas ela deve ser a mais alta possível. Além disso, quando houver uma falha, a aplicação deve retornar uma boa mensagem para o usuário, indicando o que aconteceu e o que o usuário pode fazer para contornar o problema;

- **Tolerância a falhas:** por mais confiável que a aplicação seja, sempre haverá falhas, seja porque o usuário enviou dados inválidos, fez alguma ação inadequada, entre outras causas. Uma aplicação deve saber lidar com falhas e ser recuperada para um estado válido, e nunca permitir que uma falha cause outros problemas;

- **Usabilidade:** toda aplicação tem algum modo de interação com o usuário, que pode ser no formato de telas ou via linhas de comandos. Independente da forma que for feita, a interação do usuário com a aplicação deve ser simples e fluida, de modo a facilitar o uso e o entendimento da aplicação por parte dos usuários;

- **Segurança:** praticamente toda aplicação vai coletar dados dos usuários, tais como nome, e-mail ou até mesmo dados financeiros. É obrigação da aplicação garantir a segurança dos dados dos seus usuários. Além disso, qualquer aplicação disponibilizada na internet pode ser alvo de ataques cibernéticos. Por isso, é essencial garantir a segurança da aplicação e de sua infraestrutura;

- **Monitoramento:** para garantir a confiabilidade e a segurança da aplicação, é necessário monitorar tudo o que está acontecendo durante a sua execução. Para isso, podem ser usadas diferentes ferramentas que normalmente analisam os logs das aplicações e da infraestrutura para encontrar problemas que possam estar ocorrendo.

ATRIBUTOS DE QUALIDADE

Alguns autores não gostam do nome requisito não funcional (ou *non-functional requirements*, em inglês), pois daria uma ideia negativa de que talvez esses requisitos não sejam tão importantes na implementação de uma aplicação. Por isso, é bastante comum também ver o nome **atributo de qualidade** para se referir aos mesmos conceitos como desempenho, escalabilidade e segurança.

Diversos desses requisitos são bastante relacionados entre si. Por exemplo, uma aplicação pode ter um bom *desempenho* com poucos usuários, mas, se ela não for *escalável*, o desempenho pode ficar ruim quando houver um número maior de usuários. A *elasticidade* é importante para garantir a *escalabilidade* de uma aplicação sem que sejam desperdiçados recursos computacionais (e financeiros), assim como a *tolerância a falhas* é importante para que a aplicação não pare de funcionar, o que pode afetar a *confiabilidade* da aplicação.

A pessoa arquiteta de software deve saber analisar as aplicações e fazer diversas escolhas para que todos esses requisitos sejam atendidos da melhor forma possível. Cada escolha na arquitetura de software possui vantagens e desvantagens, ou seja, algo pode ser bom para um requisito, mas pode atrapalhar outro. Normalmente essas vantagens e desvantagens podem ser pensadas em várias dimensões, como complexidade, custo, desempenho e segurança.

Por exemplo, uma escolha pode ter um desempenho melhor, mas com a execução muito mais custosa que outra; assim como uma pode ter um custo baixo, mas a complexidade para o desenvolvimento dela ser muito alta. O termo mais usado para definir essa escolha é o ***trade-off***, que, em uma tradução livre, pode ser definido como os benefícios e os problemas de possíveis escolhas a serem feitas na tomada de uma decisão.

Na arquitetura de software, algumas decisões são bem de alto nível, como escolher a linguagem de programação; o provedor de nuvem onde a aplicação será executada; o banco de dados que será utilizado; a arquitetura geral da aplicação (monólitos ou microsserviços). Outras já são de bem mais baixo nível, como decidir os dados dos quais será necessário fazer *cache* e como será implementada a comunicação com um serviço externo.

No decorrer deste livro, serão discutidos diversos tópicos que são importantes para a arquitetura de uma aplicação. Em todos eles, serão mostrados os *trade-offs* de cada decisão arquitetural feita. Diversos exemplos práticos de como implementar algumas dessas decisões serão apresentados. Os conceitos mostrados são independentes de linguagem ou tecnologia, e podem ser implementados de diversas maneiras. Neste livro, os exemplos serão implementados com a linguagem Java, a maioria utilizando o framework Spring.

Referências

IEEE. Institute of Electrical and Electronics Engineers. 1471-2000 - IEEE Recommended Practice for Architectural Description for Software-Intensive Systems. *IEEE Std. 1471-2000*, p. 1-30, 2000. Disponível em: https://ieeexplore.ieee.org/document/875998. Acesso em: 19 dez. 2023.

CLEMENTS, Paul; BACHMANN, Felix; BASS, Len; GARLAN, David; IVERS, James; LITTLE, Reed; MERSON, Paulo; NORD; Robert; STAFFORD, John. *Documenting Software Architectures*: Views and Beyond. 2. ed. Boston: Addison-Wesley Professional, 2010.

CAPÍTULO 2
Modelos de Arquitetura de Software

A arquitetura de um software pode ser definida em diversos níveis de abstração, começando no mais alto nível, onde são decididos quais componentes serão desenvolvidos e como eles se comunicam. Já em níveis mais baixos, são feitas decisões de como esses componentes serão implementados e quais tecnologias e ferramentas serão utilizadas na comunicação desses componentes.

A definição de mais alto nível normalmente é feita no início do desenvolvimento da aplicação e pode evoluir durante o uso da aplicação. Uma aplicação pode começar a ser desenvolvida em apenas um componente, utilizando uma linguagem como Java, e com o tempo ser separada em duas, uma com o front-end usando JavaScript e apenas o back-end em Java. Além disso, conforme a necessidade, o back-end pode ser dividido em vários serviços para facilitar a escalabilidade da aplicação.

Existem diversas formas de pensar em como os componentes podem ser organizados em uma aplicação, mas podemos agrupar essas formas em duas principais categorias:

- **Monólitos:** quando a aplicação toda é implementada em apenas uma unidade. Nela está todo o código, tanto o back-end quanto o front-end da aplicação. Dentro desta unidade existem várias possibilidades de organização do código, como a arquitetura em camadas e a arquitetura de microkernel;
- **Arquiteturas distribuídas:** quando a aplicação é dividida em várias unidades, ou vários serviços independentes, e esses serviços se comunicam via uma rede de computadores. Alguns exemplos desse tipo de arquitetura são a arquitetura orientada a serviços (*Service-Oriented Architecture* – SOA), a arquitetura

baseada em eventos (*Event-Driven Architecture* – EDA) e os microsserviços.

Existem várias vantagens e desvantagens em cada um dos tipos de arquiteturas. Em um monólito, por exemplo, é muito mais fácil fazer a implantação de uma aplicação, pois ele é formado por apenas uma unidade. Já em uma aplicação que usa uma arquitetura distribuída, como microsserviços, temos que implantar várias unidades, talvez em máquinas diferentes. Isso faz com que o gerenciamento para a implantação de uma grande mudança que afete vários microsserviços seja muito mais complexa do que se fosse em um monólito.

Além disso, a execução distribuída da aplicação faz com que haja muita comunicação de redes entre os serviços, o que pode fazer o desempenho da aplicação piorar e a complexidade aumentar bastante, pois a rede pode ter diversos problemas, como ter alguma lentidão que pode afetar o funcionamento de uma aplicação ou simplesmente parar de funcionar. Outra diferença grande é o monitoramento: em um monólito, temos uma única aplicação para monitorar; já com os microsserviços temos que monitorar todos os serviços.

Por outro lado, as arquiteturas distribuídas também possuem uma série de vantagens. É mais fácil dividir as equipes de desenvolvimento para trabalhar em diferentes partes do código, já que normalmente cada um dos microsserviços é praticamente uma aplicação independente. Também é muito mais fácil escalar apenas uma parte da aplicação que está tendo um grande número de requisições, em vez de ter que escalar a aplicação inteira. Além disso, com diversos serviços, a tolerância a falhas pode ser muito melhor, pois se a aplicação for bem arquitetada, não haverá um ponto único de falhas como acontece em um monólito. Nas próximas seções, vamos aprofundar mais em cada um desses tipos de arquiteturas.

2.1 Monólito

Neste modelo, as aplicações são desenvolvidas e implantadas em um único grande projeto. Pode ser uma aplicação desktop desenvolvida em C/C++ ou uma aplicação web com o código do front-end com arquivos HTML, CSS e JavaScript, e todo o código do back-end em qualquer outra linguagem (Python, Java, PHP, C/C++ etc.), tudo no mesmo repositório e distribuído em único pacote. Essa arquitetura é usada até hoje, e em muitos casos ainda é a melhor opção para o desenvolvimento de uma aplicação.

Existem muitas formas de se desenvolver um monólito. A mais simples é simplesmente criar todo o código em uma camada e tudo se comunicar livremente, já que, como tudo está no mesmo projeto, fazer isso é bastante fácil. Porém, se a aplicação for desenvolvida dessa forma, em pouco tempo fazer a sua manutenção será praticamente impossível. Por isso, mesmo um monólito deve ser bem organizado para permitir a evolução da aplicação.

Existem vários modelos para se desenvolver um monólito bem organizado e que permita a manutenção e a evolução da aplicação. Vamos discutir dois desses modelos: a *arquitetura em camadas* e o *microkernel*. A **arquitetura em camadas** divide o monólito em camadas lógicas, e normalmente cada camada se comunica apenas com outros módulos da mesma camada ou de camadas próximas (acima ou abaixo na hierarquia das camadas). Assim, uma mudança em um componente em uma camada afetará apenas poucos componentes em camadas próximas, o que facilita bastante a manutenção da aplicação.

Já o segundo modelo, o **microkernel**, é bastante usado no desenvolvimento de sistemas operacionais (SO). Nesse modelo, a aplicação possui um núcleo com seus serviços mais importantes, que controlam o estado da aplicação, e as funcionalidades para o usuário são disponibilizadas através de *plugins*, que são módulos que utilizam os serviços disponibilizados no núcleo. No exemplo do

sistema operacional, o núcleo possui as funções básicas para gerenciar o computador, tais como o sistema de arquivos, o gerenciamento de tarefas e o controle de entrada e saídas. Já as aplicações podem usar esses serviços do SO para disponibilizar funcionalidades para os usuários do computador.

Vamos discutir esses dois modelos mais detalhadamente a seguir.

Arquitetura em camadas

A arquitetura em camadas é um dos modelos de desenvolvimento de software mais usados. Nele basicamente se divide a aplicação em várias camadas, cada uma com uma responsabilidade diferente e que se complementam. Além disso, normalmente existe uma hierarquia de camadas, onde componentes de cada camada se comunicam apenas com componentes na mesma camada ou em camadas imediatamente acima ou abaixo.

No desenvolvimento web, por exemplo, é possível facilmente separar as aplicações em pelo menos três camadas: a camada de **apresentação**, a camada que implementa as **regras de negócio** da aplicação e a camada de **acesso aos dados**. A figura a seguir mostra como essa arquitetura de três camadas poderia ser organizada:

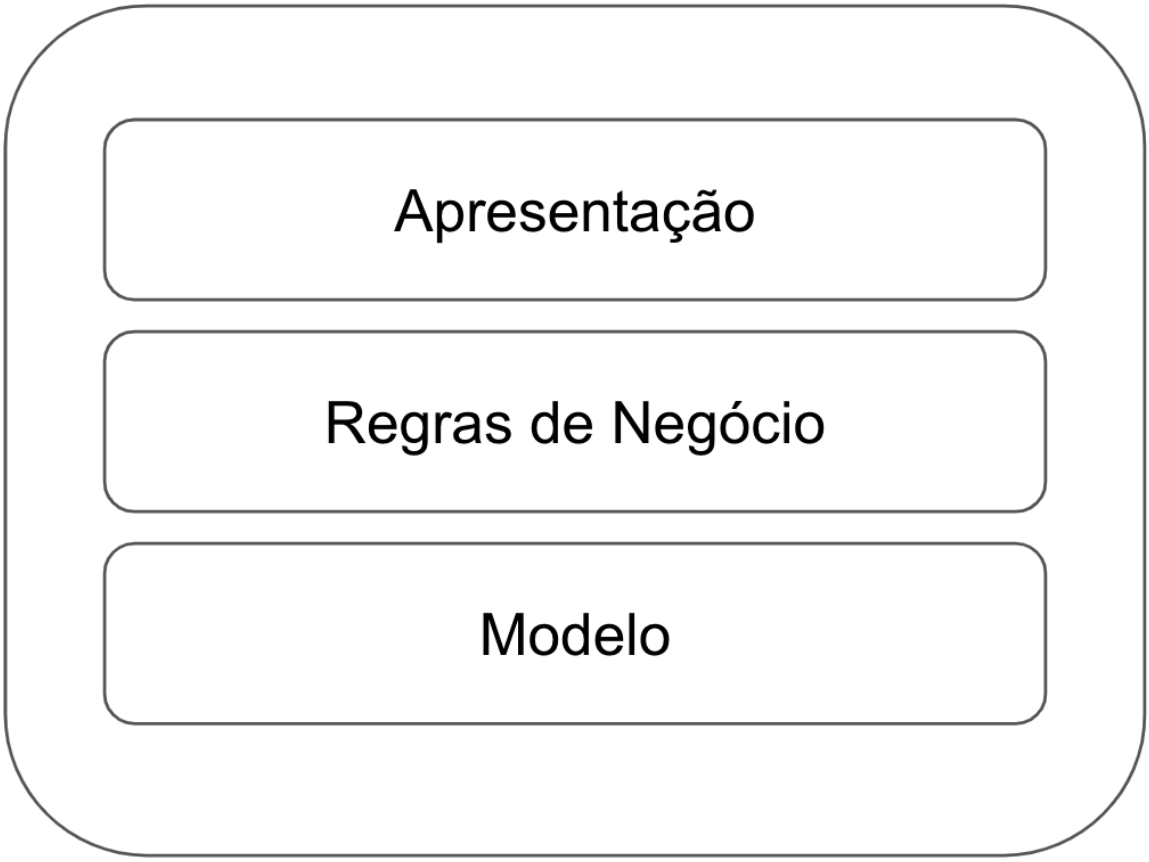


Figura 2.1: Modelo de Camadas

Neste modelo, todas as entradas e ações dos usuários são feitas na camada de *apresentação*, que tem apenas a função de validar os dados de entrada e apresentar os dados de saída. Nesta camada não deve ser feito nenhum processamento de dado, o que é responsabilidade da camada de baixo, a camada das *regras de negócio*. É ela que deve processar os dados e realizar as operações que o usuário deseja. Por fim, a camada do *modelo* é a camada que representa o modelo de dados da aplicação e que pode acessar o banco de dados.

No modelo de camadas, deve-se evitar a comunicação entre camadas que estão distantes: a camada de apresentação, por exemplo, nunca deve acessar diretamente o modelo, tudo deve passar pela camada das regras de negócio antes. Obviamente

existem variantes nesse modelo: aplicações monolíticas podem ser divididas em algumas unidades em que a camada de apresentação é desenvolvida em uma unidade separada das regras de negócio; outra divisão que é bastante comum hoje em dia é a divisão do front-end e do back-end da aplicação.

O modelo de camadas é bastante usado ainda hoje, e muitas aplicações iniciam nesse modelo até que elas atinjam um tamanho maior e comecem a ser divididas em serviços. Isso acontece pois esse modelo é extremamente simples e evita vários desafios de arquiteturas mais complexas, como os microsserviços.

Pensando nos requisitos não funcionais, esse modelo possui diversas vantagens e desvantagens. Vamos discutir algumas delas:

- **Implantação:** como a aplicação é formada apenas por um grande bloco, quando é necessário trocar uma versão, é necessário parar tudo e implantar a nova versão. Por outro lado, a implantação é mais simples, pois basta apenas trocar uma unidade de software, não sendo necessário fazer uma grande coordenação entre as atividades de implantação;
- **Desempenho:** se o modelo for bem organizado, o desempenho dele será bom, já que não existe nenhuma complexidade adicional e, por ser um monólito, não serão necessárias chamadas de rede para a execução da aplicação (exceto em chamadas a serviços externos, mas isso não é tão comum quanto em arquiteturas distribuídas);
- **Escalabilidade:** assim como no problema da implantação, por ser formada por apenas um grande bloco, mesmo que um problema de escalabilidade seja pontual em apenas uma parte da aplicação, será necessário escalar a aplicação inteira, o que pode dificultar esse processo;
- **Tolerância a falhas:** se uma falha acontecer e derrubar a aplicação, tudo parará de funcionar, já que um monólito é

apenas uma unidade de software com todas as funcionalidades implementadas;
- **Modularidade:** a aplicação ficará organizada em camadas, o que já faz uma modularização mínima da aplicação. Porém, por tudo estar na mesma base de código, é possível que os conceitos se misturem e aumentem o acoplamento das partes do código que tratam de diferentes partes da aplicação;

- **Complexidade:** esse modelo é extremamente simples de implementar, e já existem diversos frameworks que implementam um modelo de camadas simples, como o Spring MVC, que possui uma implementação bastante conhecida do modelo de três camadas usado na web, o *Model-View-Controller*;

- **Custo:** o custo de desenvolvimento, manutenção e execução desse modelo é baixo, já que é uma arquitetura simples e necessita apenas de uma máquina para a execução da aplicação. Também não há custo de rede.

Resumindo, as grandes desvantagens do modelo em camadas estão na implantação, na escalabilidade e na tolerância a falhas. Já as grandes vantagens estão no custo e na simplicidade do modelo, o que faz que esse seja um dos modelos mais utilizados para novas aplicações e MVPs (*Minimum Viable Product*).

MODELO OSI

Uma das implementações mais conhecidas do modelo de camadas é o modelo OSI (*Open Systems Interconnection*), usado em dispositivos de rede. Essa arquitetura é composta por sete camadas, cada uma com funções específicas, indo da camada física, que trata da transmissão de bits brutos, até a camada de aplicação, que fornece serviços direcionados aos aplicativos de usuário.

Microkernel

O **microkernel** também é um modelo usado em muitas aplicações, principalmente em aplicações desktop que são instaladas diretamente nas máquinas dos usuários. Nesse tipo de modelo, as aplicações são constituídas de duas partes principais: o **núcleo**, que possui todas as funcionalidades mínimas para que a aplicação seja executada, e os **plugins**, que são extensões do núcleo para prover funcionalidades mais complexas para os usuários. A figura a seguir mostra como uma aplicação que segue esse modelo pode ser implementada:

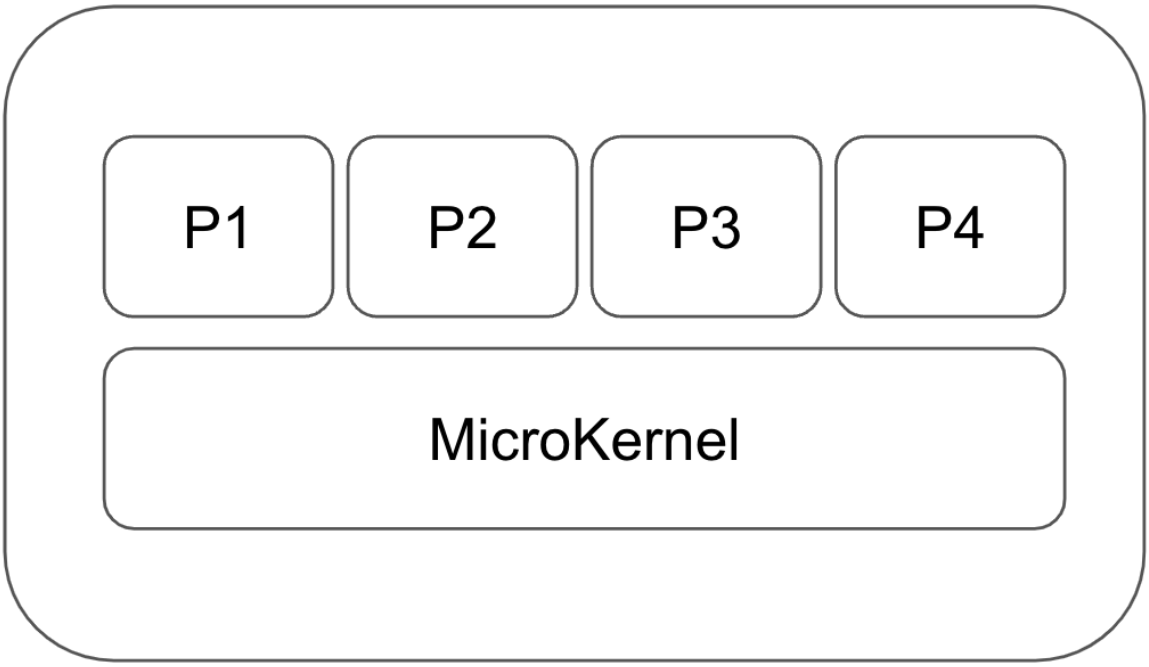


Figura 2.2: MicroKernel

Na imagem, é mostrado o componente principal, chamado de *MicroKernel*, no qual estão todas as funcionalidades básicas do sistema. Pensando no exemplo de um sistema operacional, é nesta parte que estaria implementada toda a parte de acesso a arquivos, de gerenciamento de memórias e processos e, também, o gerenciamento dos dispositivos de entrada e saída. Esses serviços

podem estar disponíveis em bibliotecas que possuem as chamadas de funções que podem acessar essas funcionalidades. Usando esses serviços, são implementados os plugins, que são as aplicações que vão usar as funcionalidades providas pelo *MicroKernel*. A figura mostra quatro, mas poderiam ser implementados muito mais plugins que utilizam um mesmo microkernel.

Um exemplo bastante conhecido de aplicação que usa esse modelo é a IDE Eclipse, que tem como principais funcionalidades no seu núcleo o gerenciamento de arquivos e projetos e o editor de arquivos (principalmente texto, mas extensível para outros formatos), e praticamente todas as outras funcionalidades são disponibilizadas como plugins. Isso torna essa ferramenta bastante extensível, sendo utilizada para diversos fins, desde o desenvolvimento de aplicações em Java até a criação de modelos usando a linguagem UML.

Em relação às vantagens e desvantagens do modelo de microkernel, elas são bem parecidas às do modelo de camadas, já que ele também é um monólito formado normalmente por apenas uma unidade de código. Duas diferenças que valem ser ressaltadas é que a escalabilidade e a tolerância a falhas, que já são um problema no modelo de camadas, são ainda mais críticas no microkernel, pois o núcleo concentra ainda mais as funcionalidades mais importantes da aplicação, o que o torna um ponto único de falha bastante importante.

2.2 Arquiteturas distribuídas

Qualquer arquitetura que divide a aplicação em diversas unidades que se comunicam em uma rede pode ser classificada como uma **arquitetura distribuída**. Existem várias formas de implementar arquiteturas distribuídas. Hoje, sem dúvida, a implementação mais comum desse tipo de arquitetura são os **microsserviços**, em que a

aplicação é dividida em vários pequenos serviços que possuem poucas ou até mesmo uma única responsabilidade. Porém existem outras arquiteturas, como a **arquitetura orientada a serviços** (*Service-Oriented Architecture* - SOA) e a **arquitetura baseada em eventos** (*Event-Driven Architecture* - EDA), que serão discutidas nas próximas seções.

Além da divisão entre serviços, uma das principais características desse tipo de arquitetura é o uso de uma rede, que pode ser a internet ou uma rede interna onde a aplicação é executada. Isso é bastante importante, pois a comunicação na rede entre os serviços afeta quase todos os requisitos não funcionais das aplicações. Por exemplo, nenhuma rede é 100% confiável, pois ela pode falhar ou apresentar lentidão; então isso deve ser considerado na confiabilidade e no desempenho da aplicação como um todo. Além disso, existem problemas de segurança que devem ser tratados na comunicação entre os serviços, pois uma rede pode ser atacada. Esse problema é bem mais simples de tratar em uma aplicação monolítica, já que nela toda a comunicação entre os componentes é interna.

No entanto, esse tipo de arquitetura tem diversas vantagens sobre arquiteturas monolíticas. Uma das principais é a facilidade de escalar os serviços. Imagine um cenário onde só uma parte da aplicação precisa de escalabilidade: em um monólito, não seria possível escalar apenas essa parte da aplicação, seria necessário escalar a aplicação inteira. Já em uma arquitetura distribuída, podemos escalar apenas a parte que terá uma demanda maior. Outra grande vantagem é que uma arquitetura baseada em serviços pode não ter um ponto central de falha, o que pode aumentar a tolerância a falhas da aplicação.

Arquitetura orientada a serviços

Alguns dos problemas dos monólitos, como a dificuldade de ter diversos times trabalhando em um mesmo código e de escalar partes específicas da aplicação, levou à ideia de separar a aplicação

em diversos serviços. Na *arquitetura orientada a serviços*, normalmente a divisão é feita usando os domínios da aplicação. Em uma aplicação de um *streaming* de vídeos, por exemplo, podemos ter um serviço para gerenciar os usuários e os pagamentos na plataforma, outro para gerenciar o catálogo dos filmes e um último apenas para controlar o envio dos vídeos para os clientes. Nesse modelo, a aplicação normalmente tem apenas um banco de dados, que é acessado por todos os serviços. A figura a seguir mostra um exemplo dessa arquitetura:

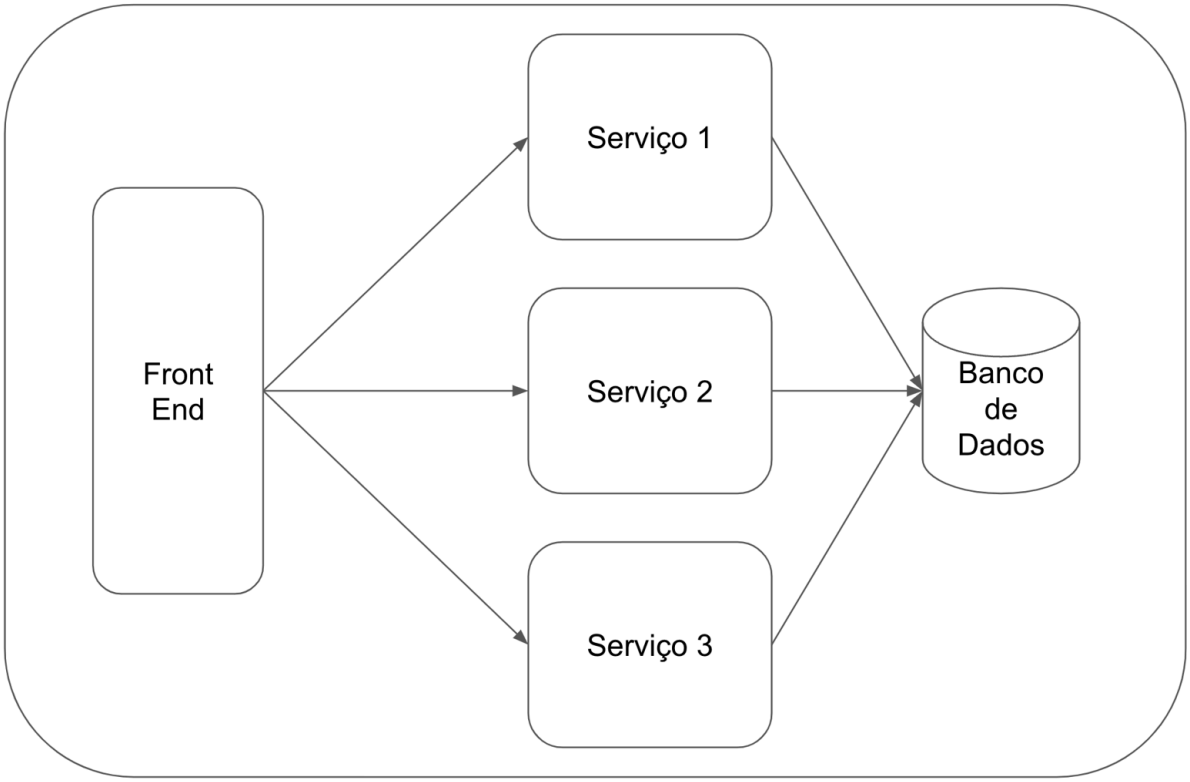


Figura 2.3: Arquitetura orientada a serviços

Nesse modelo, normalmente o front-end da aplicação acessa os serviços que disponibilizam os dados de cada um dos seus domínios. Esse modelo tem diversas vantagens sobre os monólitos. Uma delas é permitir a escalabilidade de apenas parte da aplicação. Voltando ao exemplo do *streaming* de vídeo, imagine que a parte da aplicação que envia os vídeos para os clientes está no limite, enquanto o

restante da aplicação não apresenta nenhum problema. É possível criar novas instâncias apenas dessa parte da aplicação que está apresentando problemas.

Além disso, se um serviço precisar de dados de outro domínio, ele pode acessar os dados diretamente no banco de dados. Esse acesso direto ao banco de dados tem uma grande vantagem, que é permitir o acesso a todos os dados da aplicação de forma rápida e fácil; mas também tem uma grande desvantagem, que é dificultar bastante mudanças no modelo de dados da aplicação.

Existem várias maneiras de tentar minimizar esses problemas. É possível limitar que apenas o domínio principal tenha acesso de escrita na sua parte do banco de dados, e os outros domínios, apenas de leitura. Por exemplo, na tabela *user*, apenas o serviço de gerenciamento de usuário tem acesso de escrita; os outros, apenas de leitura.

A estratégia de limitar quem escreve os dados no banco de dados até ajuda, mas ainda não resolve o problema, pois mudanças na estrutura de uma tabela podem quebrar consultas que outros serviços fazem. Por isso, a estratégia mais extrema é cada serviço ter seu banco de dados. Essa estratégia é bastante usada atualmente, principalmente na arquitetura de microsserviços que discutiremos nas próximas seções. Nela, a principal vantagem é que os banco de dados são totalmente independentes, e uma mudança na estrutura afeta apenas o serviço que acessa diretamente aquele banco de dados.

A grande desvantagem é que, se um serviço precisar de dados de um outro serviço, ele precisará fazer uma chamada de rede para buscar os dados. Por exemplo, se o serviço de catálogo de filmes precisar de informações do usuário para personalizar as indicações, ele não poderá ir direto no banco de dados, mas, sim, fazer uma chamada para uma rota do serviço de usuário que retorna as informações. Nesse exemplo, a arquitetura da aplicação seria modificada, como é mostrado na figura a seguir:

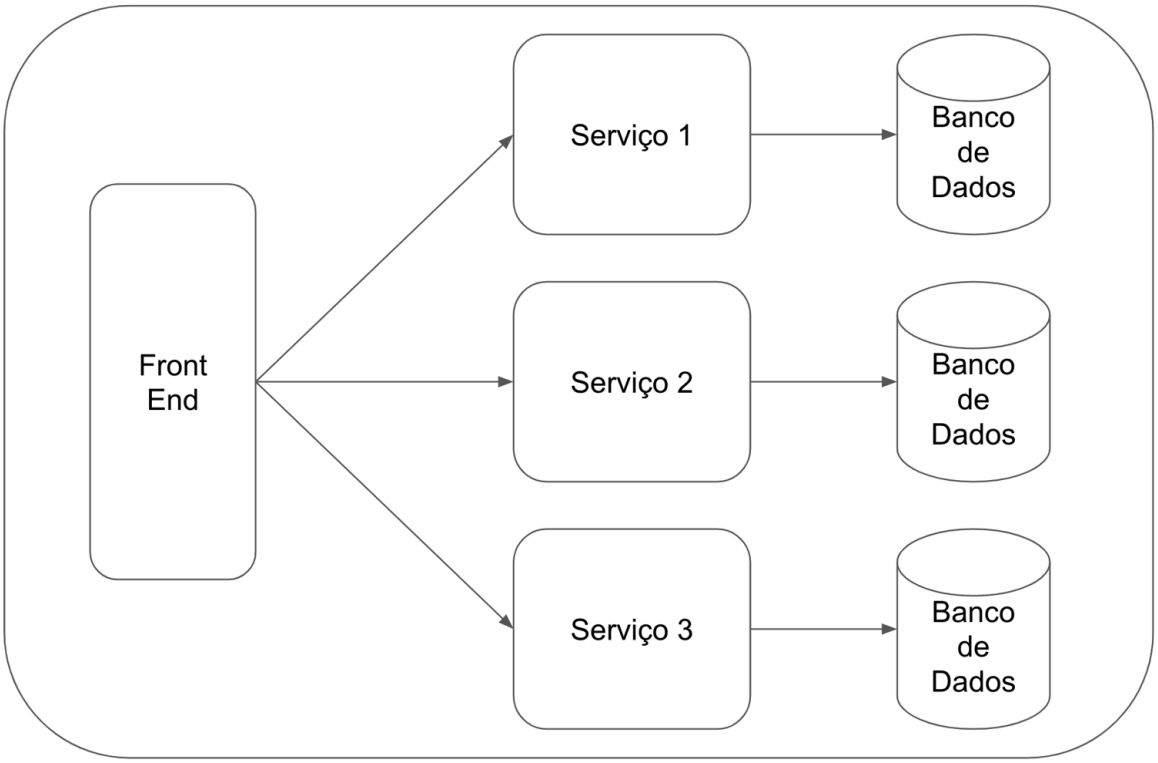


Figura 2.4: Arquitetura orientada a serviços com banco de dados segregados

As principais diferenças do primeiro modelo para o segundo é que, agora, em vez de apenas um único banco de dados para a aplicação inteira, existem vários bancos de dados, um para cada serviço. Além disso, note a seta entre os serviços: ela indica que existe uma comunicação direta entre esses serviços para a troca de informações.

Em relação aos requisitos não funcionais, as principais vantagens e desvantagens desse modelo são:

- **Implantação:** nessa arquitetura, a implantação é mais complexa do que em um monólito, mas, como o número de serviços não é tão grande, ela não costuma ser um grande problema. A grande vantagem é que as implantações dos

serviços são totalmente independentes, o que diminui o risco de erros nas atualizações dos serviços;

- **Desempenho:** o desempenho não costuma ser um problema nesse tipo de arquitetura, já que não há muita comunicação entre os serviços, o que poderia incluir atrasos por causa da rede;

- **Escalabilidade:** esta é outra grande vantagem sobre os monólitos, já que nessa arquitetura é possível escalar os serviços independentemente. Como os serviços aqui ainda são grandes, ainda terá algum desperdício por ter que escalar um serviço inteiro, mas já é muito menos custoso do que escalar um monólito;

- **Tolerância a falhas:** como os serviços são independentes, não existe um ponto único de falha. Porém, é claro que existem serviços essenciais; por exemplo, se um serviço de usuário para de funcionar, é provável que a aplicação inteira pare de funcionar;

- **Modularidade:** como cada serviço fica responsável por um domínio da aplicação, a divisão em serviços já é uma primeira etapa na modularização da aplicação. Porém, como os serviços ainda podem ser grandes, a organização interna de cada serviço ainda é bastante importante;

- **Complexidade:** em relação a um monólito, a complexidade aumenta bastante, já que são múltiplos serviços para gerenciar, monitorar e implantar. Porém, pelo número limitado de serviços, a complexidade não costuma ser um grande problema nesse tipo de arquitetura;

- **Custo:** o custo de execução é maior do que o de um monólito, mas, como o número de serviços não costuma ser grande, o custo também não é um grande problema.

Arquitetura baseada em eventos

Muitos sistemas podem ser descritos pelos vários eventos que ocorrem durante o seu uso. Em um *e-commerce*, por exemplo, podemos tratar cada passo de uma compra como um *evento*. Esses eventos podem acontecer sincronamente, isto é, gerar uma resposta assim que o evento é gerado, ou assincronamente, onde a resposta do evento pode acontecer apenas depois de um certo período de tempo.

Nesse exemplo do *e-commerce*, poderíamos ter os seguintes eventos:

- `Registrar Compra` : logo que o cliente realiza a compra, um primeiro evento é gerado indicando que uma nova compra foi realizada;
- `Validar Pagamento` : depois da compra, é preciso validar os dados do pagamento e se a compra pode ser concretizada. Se o pagamento não for autorizado, o fluxo de eventos termina aqui;
- `Separação` : quando a compra é concretizada, é necessário gerar um novo evento para que os itens da compra sejam separados e preparados para a entrega;
- `Envio` : depois de separados, os itens são enviados para a transportadora, que deve iniciar o processo de entrega do pedido;
- `Finaliza com Sucesso` : por fim, a entregadora, depois de entregar os itens, informa que o pedido foi concluído.

Esse tipo de sistema pode ser modelado por um fluxograma, como mostra a figura a seguir:

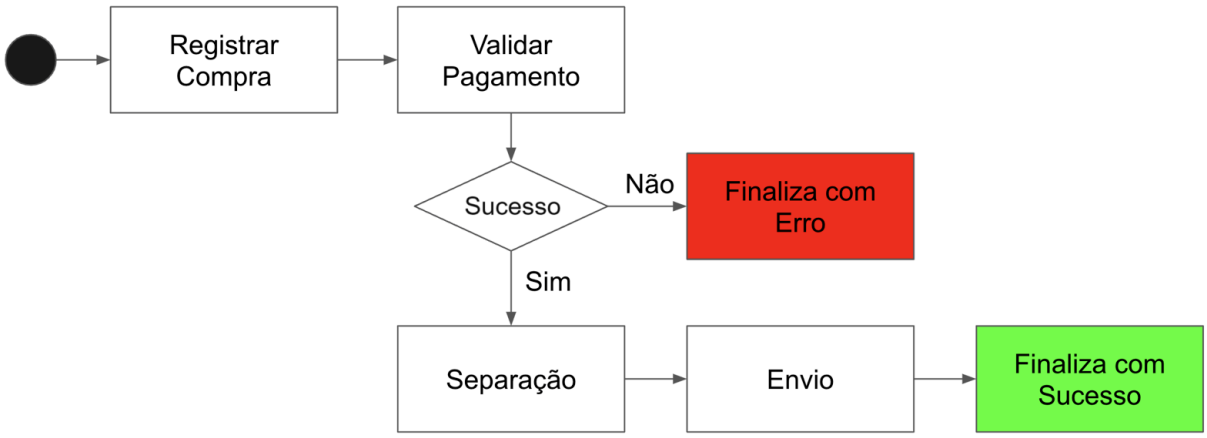


Figura 2.5: Fluxograma da aplicação de compras

Existem várias formas de implementar uma arquitetura baseada em eventos. Talvez a mais comum seja incluir um sistema que controla uma fila de eventos. Esses eventos podem ser gerados e consumidos por diferentes partes da aplicação. Normalmente, um processo que gera uma mensagem é chamado de **produtor** e o que consome, de **consumidor**. Pensando no exemplo anterior, teríamos vários produtores e consumidores.

Por exemplo, para o primeiro evento, a compra, o produtor seria o site da loja onde as compras são realizadas, e o consumidor seria um processo que faz a validação dos dados de pagamentos do cliente. Já quando a separação dos itens é concluída, o produtor seria um sistema que é executado no armazém da empresa, onde os pedidos são separados, e o consumidor é o sistema da transportadora, que recebe os dados para a realização da entrega.

Esse modelo é bastante usado atualmente, principalmente por aplicações que possuem esse modelo de máquina de estados bem claro. Ele é também usado em aplicações que precisam de um nível de escalabilidade bastante alto, já que, com a divisão clara entre vários sistemas, é possível escalar apenas as partes onde os eventos estão demorando para serem processados.

As principais vantagens e desvantagens desse modelo são:

- **Implantação:** a implantação pode ser bastante complexa, pois cada produtor/consumidor pode ser uma aplicação separada, então uma nova funcionalidade que afeta várias partes da aplicação pode demandar uma implantação grande. Por outro lado, existe uma grande vantagem: a implantação de cada parte da aplicação é separada da outra, o que diminui bastante o risco de falhas;

- **Desempenho:** o desempenho costuma ser bom em arquitetura baseada em eventos, já que a maioria das ferramentas de filas possui um excelente desempenho. Além disso, esse tipo de arquitetura normalmente é altamente paralelizável, já que os eventos são independentes uns dos outros;

- **Escalabilidade:** esse modelo é extremamente escalável, já que é possível escalar qualquer parte do sistema onde os eventos estão demorando para serem processados. Os sistemas de filas atuais, como Kafka e o RabbitMQ, também são extremamente escaláveis;

- **Tolerância a falhas:** talvez essa seja a maior vantagem desse modelo, já que, se uma aplicação cair, as mensagens continuam na fila, e assim que a aplicação voltar a funcionar, os eventos voltam a serem processados;

- **Modularidade:** existem várias formas de implementar essa arquitetura, mas, como normalmente ela é dividida em partes específicas que processam cada um dos eventos que a aplicação gera, na maioria dos casos ela já tem pelo menos um nível de modularização;

- **Complexidade:** uma das maiores desvantagens desse modelo é a sua complexidade. Dependendo do tamanho e da quantidade de eventos que são gerados, conseguir depurar e entender o que está acontecendo na aplicação pode ser bastante difícil;

- **Custo:** o custo de execução desse modelo pode ser bastante alto, já que podem existir vários serviços sendo executados, o que demandará uma infraestrutura grande para executar a aplicação. Além disso, por esse modelo ser fortemente baseado na troca de mensagens, o uso da rede de comunicação também será bastante grande, o que pode aumentar os custos de execução da aplicação.

Microsserviços

Talvez a arquitetura de software mais comentada e mais usada atualmente é a de **microsserviços**. A ideia geral é parecida com a da arquitetura orientada a serviços, mas com uma divisão entre os serviços muito mais drástica. Por exemplo, na arquitetura orientada a serviços, pode existir um serviço que trata todo o processo de compras de um *e-commerce* contendo as entidades para esse processo, como as que representam as compras, a entrega e o meio de pagamento. Já com microsserviços, a divisão é muito maior: nesse mesmo exemplo, poderíamos ter um serviço apenas para receber as compras, outro para processar os pagamentos e outro que efetua a entrega.

A figura a seguir mostra como uma arquitetura de microsserviços pode ser implementada:

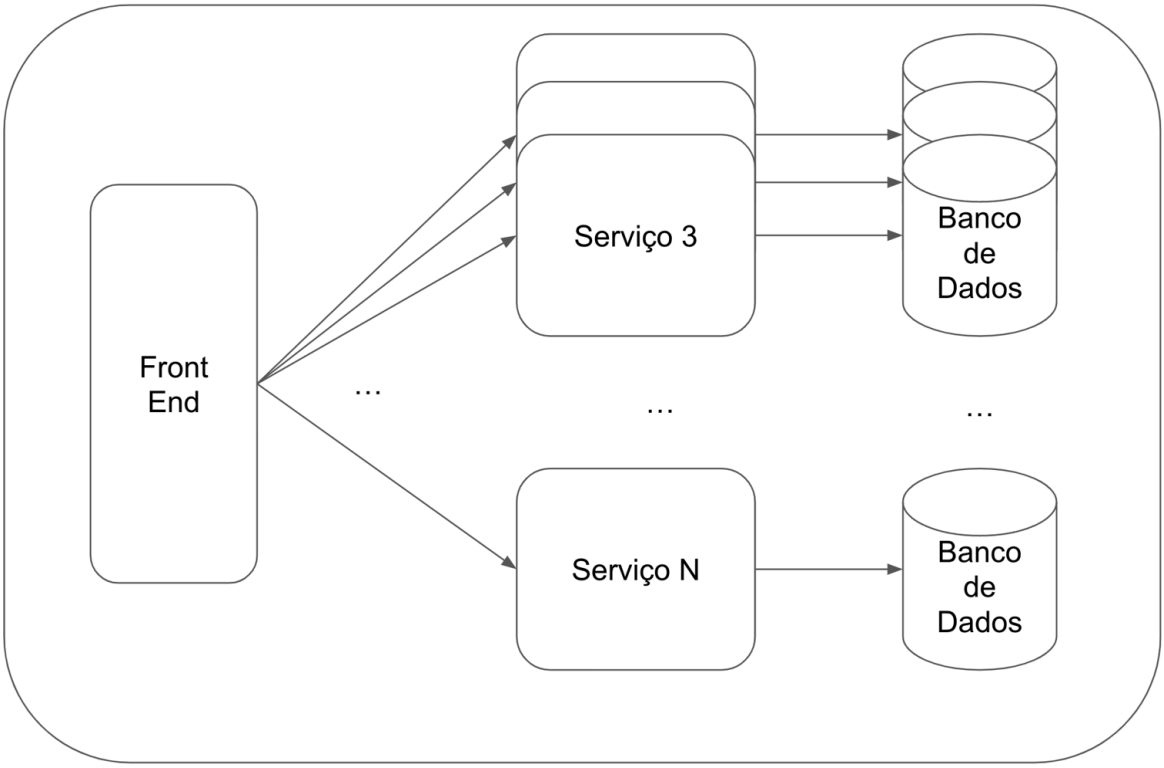


Figura 2.6: Arquitetura de microsserviços

O objetivo dessa arquitetura é manter os serviços os menores possíveis, o que facilita muito o entendimento do projeto e a sua manutenção e evolução. Há duas grandes desvantagens desse modelo: a primeira é a complexidade de gerenciar os muitos serviços que devem funcionar conjuntamente; a segunda é a grande dependência da rede, pois a comunicação entre serviços é muito usada, já que para realizar uma tarefa simples pode ser necessária a execução de diversos serviços que, por sua vez, estão executando em diferentes processos.

Vamos pegar como exemplo uma aplicação de entrega de refeições. Ela poderia ser dividida nos seguintes serviços:

- `Usuários` : gerencia os usuários que estão cadastrados na plataforma, permitindo a adição e exclusão de contas e o gerenciamento de dados do usuário, como meios de pagamentos e endereços;

- `Restaurantes` : gerencia os restaurantes cadastrados na plataforma, permitindo a adição, exclusão e edição de restaurantes e, também, do menu desses restaurantes;

- `Entregadores` : gerencia os entregadores cadastrados na plataforma. Também deve possuir um serviço que consegue encontrar o melhor entregador para um pedido específico considerando fatores como o horário, o endereço e o tipo da entrega;

- `Pedidos` : possui rotas para o recebimento e o acompanhamento de pedidos. Quando um pedido é realizado, deve validar os dados do pedido, como os itens que foram comprados e os dados dos usuários da plataforma. Quando o pedido é confirmado, deve enviar os dados para o serviço de pagamento;

- `Pagamento` : recebe os dados de um pedido confirmado e realiza o pagamento, normalmente integrando com um serviço externo que processa o pagamento. Caso o pagamento seja confirmado, passa para a próxima etapa, que é a entrega do produto;

- `Entrega` : depois que o pedido teve o pagamento confirmado, é preciso buscar um entregador que esteja disponível para fazer a entrega; além disso, é preciso buscar os dados de entrega do cliente e, com isso, criar o fluxo de entrega até que o entregador confirme que o pedido foi entregue.

Dependendo do caso, esses serviços poderiam ser divididos em unidades ainda menores. Por exemplo, em um caso extremo, poderíamos ter um microsserviço que apenas lista os restaurantes que estão em algum cache. Novos restaurantes, porém, são cadastrados em um outro microsserviço e, depois de algum tempo, o cache do outro microsserviço é atualizado. Essa divisão depende de vários fatores: no nosso exemplo, se a busca de restaurante é uma funcionalidade crucial que deve ser escalada sozinha, talvez valha a pena a complexidade de ter um serviço novo.

O fluxo de comunicação entre os microsserviços pode ser implementado de várias formas: os serviços podem se comunicar diretamente para avisar que cada um terminou sua parte do processamento; ou pode existir um processo centralizado que funciona como um orquestrador e controla o fluxo entre todos os serviços; ou, ainda, pode existir uma fila de eventos que é usada por vários serviços (é bem comum microsserviços usarem técnicas da arquitetura baseada em eventos). A comunicação entre esses serviços pode ser de forma síncrona, normalmente usando chamadas REST, ou de forma assíncrona, usando alguma ferramenta de mensageria como o Kafka ou o RabbitMQ.

Existem ainda outras variações nesta arquitetura. Um padrão bastante utilizado em microsserviços é o `api-gateway`, que é uma camada que controla o acesso à API e que pode ter diversas funcionalidades. Por exemplo, no primeiro esquema, em que o front-end acessava diretamente os serviços, a aplicação do front-end tinha que conhecer todos os serviços disponíveis; já com o `api-gateway`, o front-end só precisa conhecer o *gateway*, e este é o responsável por distribuir as chamadas para os serviços certo.

Além disso, existem diversas funcionalidades que poderiam ser replicadas nos serviços, como a limitação de requisições e a autenticação. Isso pode ser implementado diretamente no *gateway*, evitando a repetição de código e de funcionalidades nos serviços.

A figura a seguir mostra como fica a arquitetura com um `api-gateway`:

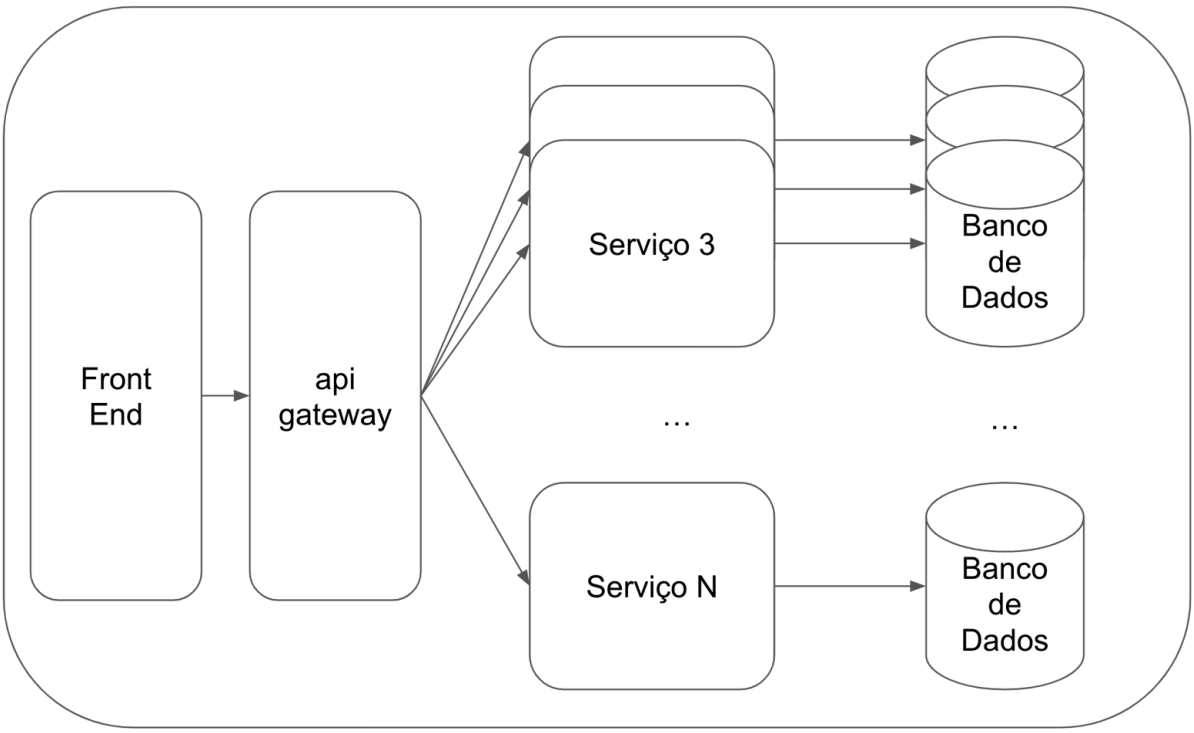


Figura 2.7: Arquitetura de microsserviços com um api-gateway

As vantagens e desvantagens desse modelo são bem parecidas com as da arquitetura baseada em eventos, pois a divisão em vários serviços também traz os mesmos benefícios da arquitetura anterior. Algumas diferenças são:

- **Implantação:** uma das grandes vantagens dos microsserviços é a implantação separada de cada um dos serviços. Pensando no exemplo do sistema de entrega, se houver a necessidade de implantação de uma nova versão do serviço de entrega, essa atividade não terá nenhuma interferência nos outros serviços, o que facilita o trabalho e diminui muito o risco de problemas que afetem a aplicação inteira;
- **Desempenho:** o desempenho em microsserviços pode não ser bom, já que existe muita comunicação de rede, o que pode adicionar atrasos na execução dos serviços;
- **Escalabilidade:** como os serviços normalmente são pequenos, é possível escalar apenas a funcionalidade que necessita

atender a um maior número de requisições;

- **Tolerância a falhas:** também devido ao grande número e da grande especialização dos serviços, a tolerância a falhas é uma grande vantagem dessa arquitetura;
- **Modularidade:** devido à grande divisão de serviços, a modularidade é quase natural nos microsserviços. Obviamente existem desafios de como dividir e como não repetir código em diversos serviços, mas, normalmente, se bem implementada, a arquitetura de microsserviços é bastante modular;
- **Complexidade:** a quantidade de microsserviços (aplicações independentes) que pode existir é muito grande, por isso a complexidade de gerenciar todos os serviços, incluindo tarefas como implantação, monitoramento e depuração, é bastante grande;
- **Custo:** o custo de execução pode ser bastante alto, já que será necessário executar muitos serviços independentes e ter diferentes bancos de dados para executar a aplicação toda. Além disso, pode existir o custo da rede, pois haverá bastante comunicação entre os serviços.

Arquitetura hexagonal

Um microsserviço pode ser organizado internamente de várias formas, utilizando, por exemplo, a arquitetura em camadas ou um microkernel. Uma forma de organização que ficou bastante conhecida e vem sendo bastante utilizada é a **arquitetura hexagonal** (que também pode ser usada em diferentes contextos, como em um monólito). Nessa arquitetura, a ideia é concentrar a parte específica da aplicação em um núcleo, que contém uma camada com as entidades que representam o modelo de dados da aplicação e uma camada que implementa as regras de negócio da aplicação.

Conectados a esse núcleo, existem os adaptadores de entrada e saída. Os adaptadores de entrada são as formas com que a

aplicação se conecta a aplicações que utilizam os seus serviços, como uma API REST. Já os adaptadores de saída conectam-se a ferramentas utilizadas internamente pela aplicação, como bancos de dados, caches e APIs de serviços externos.

A figura a seguir mostra como a arquitetura hexagonal pode ser implementada:

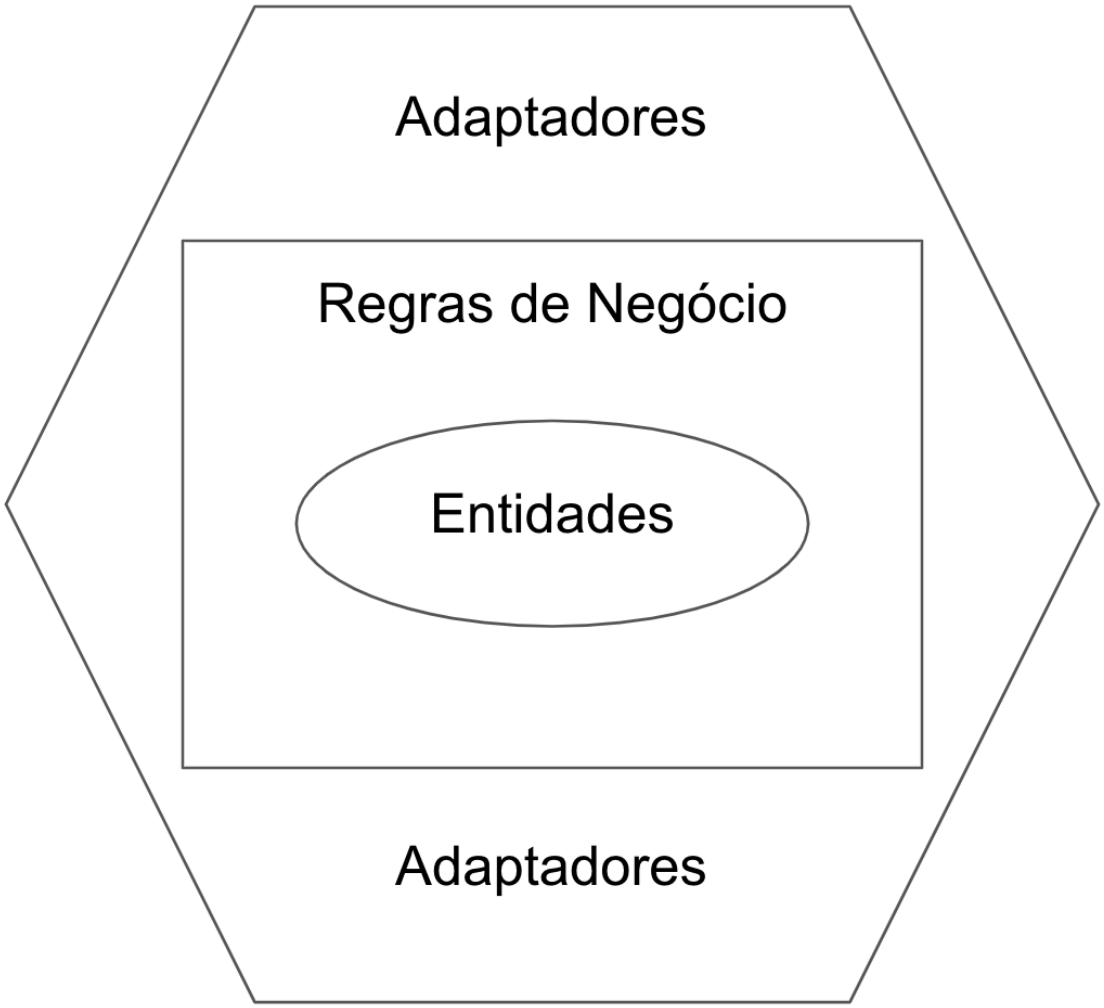


Figura 2.8: Arquitetura hexagonal

Esse tipo de arquitetura pode ser bastante útil para aplicações que possuem um grande número de interações com ferramentas externas, seja usando adaptadores de entrada quanto de saída. Ela também pode ser útil para aplicações que podem utilizar mais de uma tecnologia para um mesmo fim, como dois bancos de dados diferentes ou APIs REST ou GRPC. Isso acontece porque ela reforça a separação das regras de negócio e entidades da aplicação das integrações que são feitas com a aplicação, e para cada nova tecnologia que será integrada à aplicação, basta implementar um novo adaptador.

Conclusões

Existem muitos modelos de arquitetura para a aplicação, e é impossível dizer que um modelo é superior a outro. O que podemos dizer com certeza é que existem modelos que se adaptam melhor a cada necessidade de um projeto. Se a aplicação ainda está em um estágio inicial, sendo desenvolvida como um MVP para uma startup, talvez um monólito bastante simples, que utilize um modelo de camadas que separe o front-end e o back-end da aplicação e modularize bem os componentes internos, seja a melhor opção, pois esse modelo será rápido de desenvolver e barato de ser mantido no início.

Tentar iniciar essa aplicação já usando uma arquitetura supercomplexa, com dezenas de microsserviços, pode ser um grande erro, pois a execução será cara e talvez o domínio para a divisão dos serviços ainda não esteja tão claro, o que fará com que sejam necessárias várias refatorações nos serviços. Porém, quando a aplicação já estiver madura, começar a ter muitos usuários e apresentar alguns gargalos, talvez seja a hora de começar a dividi-la em serviços menores, que sejam mais fáceis de manter e de escalar quando necessário. Depois, talvez fique claro um fluxo que faça bastante sentido implementar usando uma arquitetura baseada em eventos.

Por isso, a arquitetura da aplicação não é uma coisa estática. Assim como uma aplicação pode receber melhorias e novas funcionalidades com o tempo, a sua arquitetura também pode evoluir e as necessidades irem mudando com o tempo. No início, talvez o mais importante seja garantir a confiabilidade da aplicação e, depois de um tempo, com o aumento no número de usuários, a escalabilidade passa a ser o foco principal. Além disso, depois de começar a ficar conhecida, a aplicação provavelmente vai ser alvo de mais tentativas de ataques, e o foco em segurança precisará aumentar.

Parte 2: Desempenho

Capítulo 3
Desempenho

Um dos problemas mais comuns na Engenharia de Software é garantir que uma aplicação mantenha um bom desempenho quando a quantidade de dados que ela processa aumenta. Normalmente medimos o desempenho de um trecho de código pensando no seu *tempo de execução*. Por exemplo, em uma API REST, podemos medir o desempenho pensando no tempo de resposta de cada uma das rotas dessa API. O tempo depende bastante do código que está sendo executado, por isso não existe um tempo de resposta padrão para dizermos se o desempenho desse código é bom ou ruim.

Em um e-commerce, uma rota simples que retorna detalhes de um produto deve ser bastante rápida; se ela demorar mais de 1 segundo para executar, possivelmente temos algum problema de desempenho. Por outro lado, uma rota para finalizar a compra, que precisa se comunicar com outro serviço para a efetivação de pagamento, pode demorar um tempo um pouco maior. Então, o desempenho é bastante dependente do processo que será executado.

Vários fatores podem influenciar o tempo de execução de um trecho de código, e um deles é a própria implementação do código. Existem diversos algoritmos que podem levar um tempo de execução maior para uma mesma funcionalidade – por exemplo, a ordenação de uma lista. Outro fator que influencia bastante o desempenho de um código é o acesso a recursos externos, como acesso a arquivos, ao banco de dados ou a outro serviço na internet.

Outro requisito não funcional que está bastante relacionado ao desempenho é a *escalabilidade*. Independente do tipo da aplicação, sempre existirá um limite de execuções que ela consegue realizar

em um determinado período de tempo. Talvez a forma mais simples de exemplificar isso seja através de uma aplicação web, que é executada em um servidor. Neste caso, diversos clientes podem acessar essa aplicação concorrentemente – a figura a seguir ilustra a arquitetura em alto nível dessa aplicação:

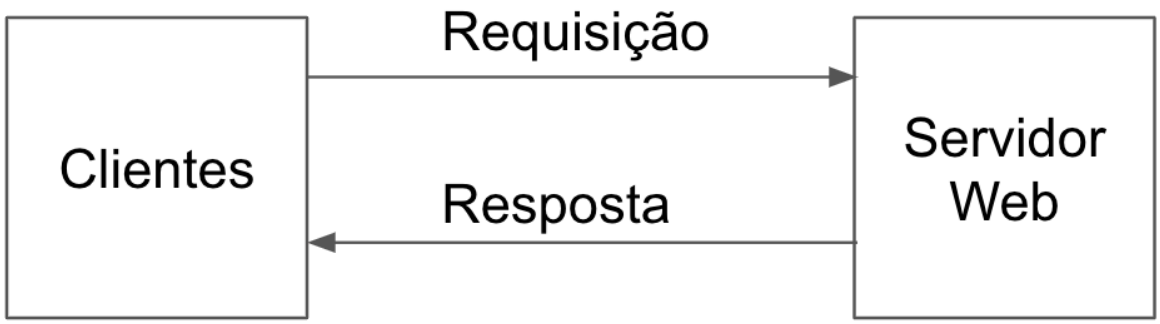


Figura 3.1: Aplicação web que atende requisições

Essa aplicação terá um bom desempenho enquanto o número de requisições — isto é, o número de clientes que estão tentando acessar a aplicação — ficar dentro do que o servidor suporta. Vários fatores podem influenciar esse limite: por exemplo, o hardware do servidor onde a aplicação está sendo executada pode aumentar ou diminuir esse limite de requisições; se uma máquina possui diversos processadores, ela vai conseguir atender a um número maior de requisições, pois o nível de paralelismo nela vai ser maior.

Então, para melhorar o desempenho de uma aplicação, basta melhorar o hardware onde ela está sendo executada? Se a resposta fosse *sim* sempre, não precisaríamos pensar muito em arquitetura de software. Porém, a resposta é: *depende*. Sem dúvida, em alguns casos basta melhorar um pouco o servidor, aumentando a memória e o número de CPUs. No entanto, se a aplicação continuar a ter um número crescente de requisições, entraremos em um ciclo em que sempre será necessário melhorar o hardware.

A melhoria no hardware é limitada, pois o custo para a execução da aplicação também fica cada vez mais alto, não só pelo preço das máquinas, mas também pelos custos recorrentes — máquinas melhores normalmente também possuem custos de manutenção maiores, consomem mais energia e precisam de sistemas de resfriamento melhores (que são mais caros e gastam mais energia). Por isso, antes de simplesmente comprar máquinas cada vez melhores, precisamos também pensar em *técnicas* que possam melhorar o desempenho da aplicação, independente do hardware em que ela esteja sendo executada. Essas técnicas podem estar tanto no código da aplicação quanto na infraestrutura em que as aplicações são executadas. Dois exemplos de técnicas que podem ser utilizadas são:

- **Cache:** praticamente todo serviço precisa acessar uma fonte de dados externa, como um banco de dados ou um arquivo no sistema, e esses acessos normalmente são lentos. Se esses dados raramente mudam e não ocupam um espaço grande em memória, talvez valha a pena mantê-los em memória na própria aplicação. Esse tipo de implementação é chamado de *cache*. Veremos exemplos de como implementar um cache no próximo capítulo.

- **Balanceamento de carga:** em vez de melhorar o hardware de um servidor, você pode criar diversas instâncias de uma aplicação e usar uma tecnologia chamada **balanceador de carga** (*Load Balancer*) para distribuir a carga entre essas diversas instâncias, o que melhora o desempenho e também aumenta a confiabilidade de uma aplicação. Discutiremos o balanceamento de carga no capítulo 5, *Desempenho e escalabilidade na infraestrutura*.

Essas duas tecnologias são apenas exemplos do que pode ser feito para melhorar o desempenho e a escalabilidade de uma aplicação, mas existem diversas outras técnicas. Obviamente a utilização dessas técnicas podem melhorar o desempenho da aplicação, mas elas também trazem diversas outras complexidades para a aplicação

e, por isso, deve-se conhecer bem as vantagens e desvantagens que o uso de uma tecnologia pode trazer.

Nos próximos dois capítulos veremos algumas das técnicas existentes e suas vantagens e desvantagens, mas, antes de começar a discutir as técnicas que podemos utilizar, é importante entender alguns dos principais requisitos não funcionais que estão ligados ao desempenho da aplicação, como a *escalabilidade* e a *elasticidade*.

3.1 Escalabilidade

Inicialmente, uma aplicação pode ter um pequeno número de usuários, e nada especial precisará ser feito para atender a essa demanda. Porém, caso essa aplicação se torne famosa, ela começará a ter um grande número de acessos. Dizemos que um sistema é **escalável** quando ele consegue atender a essa crescente demanda sem que o desempenho da aplicação seja afetado.

Garantir que um sistema seja escalável pode se tornar um grande desafio, dependendo do tipo de sistema que estamos desenvolvendo. Por exemplo, uma rede social precisa ser capaz de receber uma grande quantidade de dados e exibir conteúdos relevantes para um grande número de usuários. Provavelmente uma simples arquitetura cliente-servidor, com apenas um servidor para atender todas as requisições, não vai suportar a carga por muito tempo.

Ainda nesse exemplo, conforme a quantidade de usuários da rede social cresce, será necessário implementar diversas técnicas para que o sistema consiga atender toda a demanda. Inicialmente pode ser que apenas melhorar o hardware inicial que executa o sistema seja o suficiente, mas, depois de um tempo, os custos com hardware podem ficar muito altos. Será necessário, então, fazer melhorias no código da aplicação para que a sua eficiência seja maior, e talvez

seja necessário adicionar novos componentes na aplicação, como sistemas de cache e balanceadores de carga.

Existem dois tipos principais de escalabilidade: a *vertical* e a *horizontal*. Na **escalabilidade vertical**, a ideia é aperfeiçoar o desempenho da aplicação, seja pela melhoria do hardware que executa a aplicação, por meio de uma melhor implementação do sistema – com a implementação de algoritmos mais eficientes ou linguagens mais rápidas – ou pelo desenvolvimento de melhorias na arquitetura da aplicação. A figura a seguir mostra um exemplo de como pode ser feita a escalabilidade vertical.

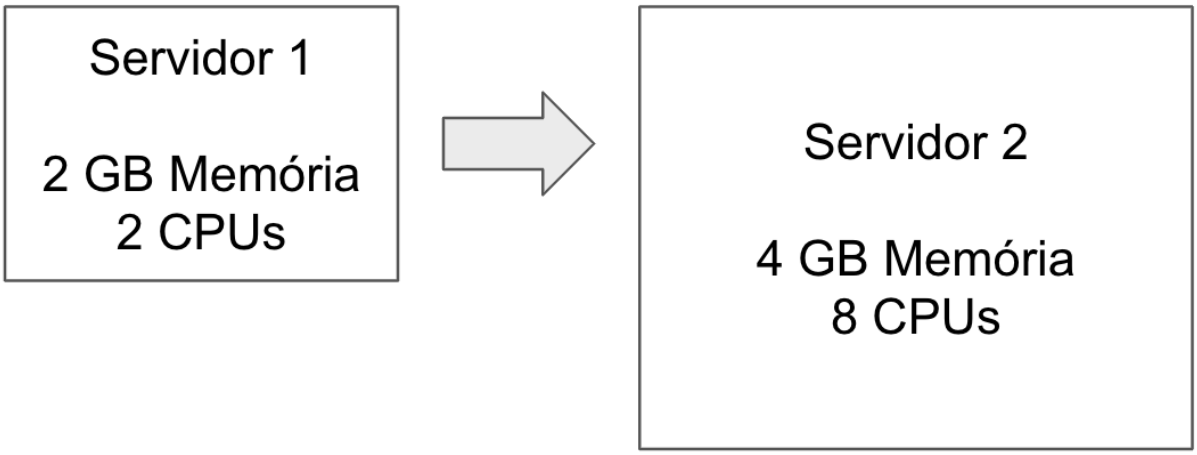


Figura 3.2: Escalabilidade vertical

Já na **escalabilidade horizontal**, o desempenho da aplicação é melhorado pela adição de recursos computacionais que executam aquela aplicação. Nesse tipo de escalabilidade, normalmente é necessário adicionar um *balanceador de carga* (*Load Balancer*), que distribui as requisições entre as várias instâncias de uma aplicação. A figura a seguir mostra um exemplo de como pode ser feita a escalabilidade horizontal:

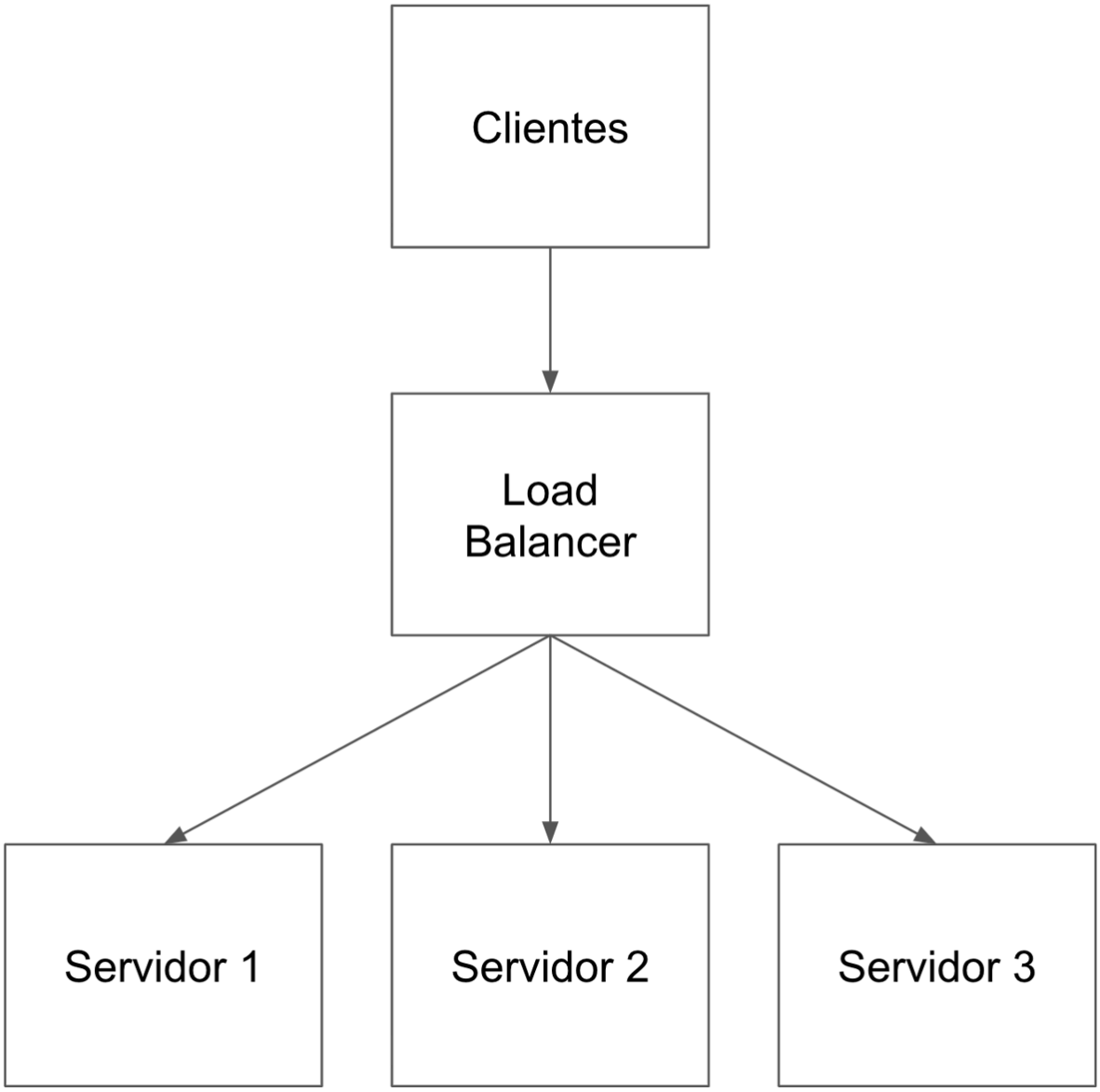


Figura 3.3: Escalabilidade horizontal

3.2 Elasticidade

Em aplicações modernas que são executadas na internet, um dos problemas mais comuns é a *flutuação de demanda*. Por exemplo, e-commerces têm uma demanda muito mais alta perto de datas comemorativas como o Natal ou o Dia das Mães; aplicações de

streaming de vídeo têm mais usuários durante fins de semana, e aplicações bancárias têm mais acessos em períodos próximos a datas de pagamento.

Se durante essas flutuações de demanda a quantidade de recursos de uma aplicação não for adequada, teremos dois possíveis problemas: ou os recursos serão insuficientes para atender a demanda, e a aplicação terá problemas de desempenho; ou os recursos ficarão ociosos, o que gera gastos desnecessários. Dizemos que um sistema tem **elasticidade** quando ele consegue equilibrar a quantidade de recursos computacionais para que a aplicação não tenha problemas de desempenho nem desperdice recursos computacionais.

Existem diversas tecnologias atualmente que aumentam ou diminuem os recursos utilizados automaticamente. Por exemplo, em um cluster que usa uma ferramenta como o Kubernetes, podemos definir um número mínimo e máximo de máquinas que vão executar aquela aplicação. A partir desses números, essa ferramenta monitora diversos recursos das máquinas, como CPU e memória, e, dependendo de alguns limites definidos pelo administrador do cluster, como utilização de memória e de CPU, aumenta ou diminui o número de instâncias.

Vamos ver um exemplo de um cenário em que esse tipo de tecnologia pode ser bastante útil. Imagine uma aplicação de entrega de refeições. Quando o dia inicia, durante a madrugada ainda, poucos pedidos são feitos, então com poucas máquinas a demanda consegue ser atendida; porém, perto do horário do café da manhã, o número de pedidos já aumenta, então mais máquinas são necessárias. Durante o horário do almoço e do jantar, a demanda aumenta ainda mais, então ainda mais máquinas serão necessárias. A figura a seguir traz um gráfico que mostra como poderia ser a variação do número de máquinas utilizadas ao longo de um dia:

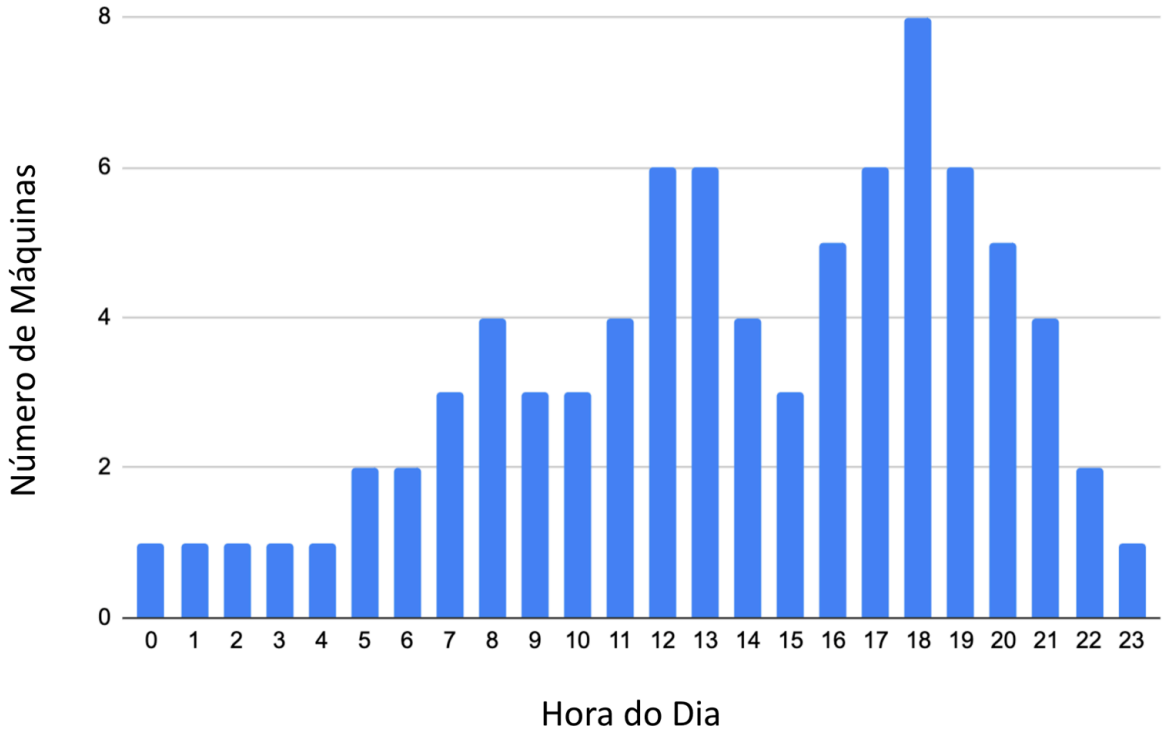


Figura 3.4: Variação do número de máquinas utilizadas durante o dia por uma aplicação

Note que a imagem mostra uma grande flutuação na quantidade de máquinas que estão sendo usadas. Se não existe uma tecnologia como essa, seria necessário manter as 8 máquinas (que é o máximo usado durante o pico do jantar) ligadas o tempo todo. Isso aumentaria muito o custo da execução da aplicação. Em ambientes de nuvem como a AWS, onde a execução das máquinas é cobrada por hora, fica bem fácil de ver a grande economia que a elasticidade pode trazer para a execução das aplicações.

Conclusões

Pensar nos três requisitos não funcionais comentados neste capítulo – desempenho, escalabilidade e elasticidade – é essencial para qualquer aplicação que terá um número significativo de usuários. É papel da pessoa arquiteta de software definir as técnicas e padrões que serão utilizados para garantir que a aplicação tenha um bom

tempo de resposta e suporte a demanda exigida sem que o custo de execução seja muito alto.

Nos próximos dois próximos capítulos, discutiremos diversas técnicas que podem ser utilizadas para ajudar as aplicações a terem um bom desempenho. No próximo capítulo, veremos algumas técnicas que podem ser implementadas diretamente *na aplicação*, como implementação de melhores algoritmos, cache e índices; no capítulo seguinte, veremos melhorias que podem ser feitas *na infraestrutura*, como o balanceamento de carga.

Capítulo 4
Desempenho e escalabilidade no código

Para entender como o desempenho e a escalabilidade de uma aplicação podem ser melhorados diretamente no código, vamos partir de um exemplo. Imagine que está disponível um novo sistema de pagamento que tem as mesmas características do PIX oferecido pelo Banco Central do Brasil. Inicialmente poucos usuários usarão esse novo sistema de pagamento, mas, com o passar do tempo, o número de usuários pode crescer exponencialmente e a aplicação começar a ter problemas de desempenho.

Algumas das possíveis causas de problemas de desempenho são: más escolhas em alguns algoritmos de processamento de dados; acessos repetidos a recursos externos; ou falta de índices no acesso aos dados. Antes de simplesmente partir para a melhoria de hardware, podemos aplicar diversas técnicas diretamente no código que podem resolver alguns dos problemas citados anteriormente, fazendo com que a escalabilidade do sistema aumente sem aumentar os custos de infraestrutura da aplicação.

Neste capítulo, vamos estudar alguns dos problemas que podemos encontrar no nosso código e algumas técnicas que podemos utilizar para resolver esses problemas.

4.1 Melhorias na implementação

Muitas vezes, uma simples melhoria na implementação pode aumentar bastante o desempenho e a escalabilidade de uma aplicação. Você já deve ter ouvido falar, por exemplo, em complexidade de algoritmos: isso pode influenciar muito o funcionamento de uma parte do código de uma aplicação. Um

exemplo bem simples é quando precisamos iterar sobre duas listas, fazendo dois comandos `for` aninhados.

Vamos implementar um exemplo e fazer alguns testes para verificar a diferença no desempenho e na escalabilidade das soluções. Imagine que temos duas listas, uma com todos os usuários cadastrados no sistema e outra com todos os pagamentos realizados em um determinado período, e temos que implementar uma rota REST que retorna uma lista com o somatório de todos os pagamentos feitos por todos os usuários do sistema. O primeiro passo desta implementação são as classes do modelo, as classes `Cliente` e `Pagamento`:

```
public class Cliente {
    private String nome;
    private String cpf;
    private Integer totalPagamentos;
}

public class Pagamento {
    private String cpf;
    private Integer valor;
}
```

Agora precisamos implementar a rota REST para calcular o total dos pagamentos para cada um dos clientes. Esse dado será salvo no campo `totalPagamento` da classe `Cliente`. Uma maneira de implementar isso é demostrada no código a seguir:

```
@GetMapping
public List<Cliente> getValue() {

    for (Cliente cliente : clientes) {
        for (Pagamento  pagamento : pagamentos) {
            if (pagamento.getCpf().equals(cliente.getCpf())) {
                cliente.setTotalPagamentos(
                    cliente.getTotalPagamentos()
                    + pagamento.getValor());
            }
```

```
            }
        }
        return clientes;

    }
```

Nessa implementação, iteramos sobre a lista de clientes e, para cada cliente, iteramos pela lista de pagamentos, verificando se aquele pagamento foi feito por um determinado usuário. Se sim, adicionamos o valor do pagamento para aquele usuário.

Essa implementação tem uma complexidade quadrática (n^2), isto é, o tempo de execução é o quadrado do número de elementos (*n*) que existem nas listas. Então, o tempo de execução da aplicação escalará mais ou menos como o do gráfico a seguir:

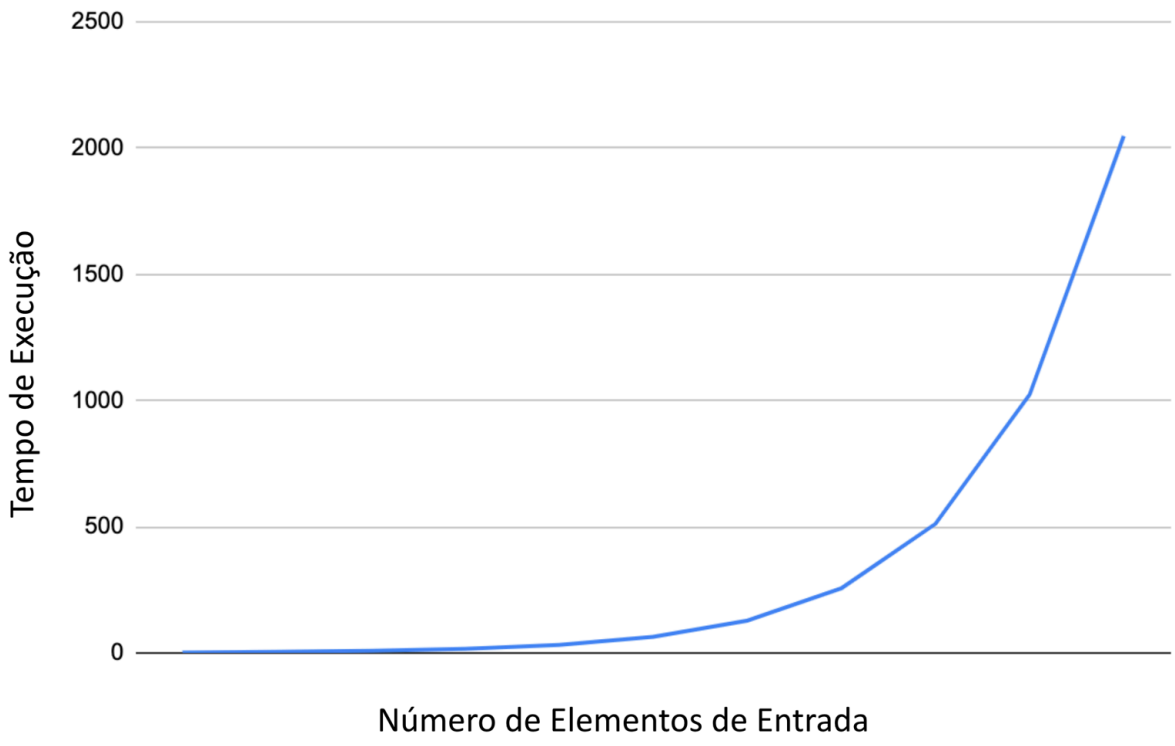


Figura 4.1: Complexidade quadrática — Tempo de execução x Número de elementos de entrada

Outra implementação possível para o mesmo método é, em vez de dois comandos `for` aninhados, utilizar uma estrutura de dados como

um *HashMap* do Java ou um dicionário do Python. Primeiro criamos um `for` apenas para calcular o somatório de pagamentos dos usuários, e depois fazemos outro `for` para encontrar a lista de pagamentos realizados para cada usuário que foi salvo no `HashMap`.

```
@GetMapping
public List<Cliente>  getValue() {

    Map<String, Integer> clientesPagamentos = new HashMap<>();
    for (Pagamento  pagamento : pagamentos) {
        clientesPagamentos.merge(pagamento.getCpf(),
                pagamento.getValor(), Integer::sum);
    }

    for (Cliente cliente : clientes) {
        Integer valor = clientesPagamentos
            .get(cliente.getCpf());
        cliente.setTotalPagamentos(valor);
    }
    return clientes;

}
```

Essa implementação, além de muito mais simples, tem uma complexidade linear (*n*), isto é, o tempo de execução cresce linearmente com o número de elementos na lista. O tempo de execução nesse tipo de implementação seguirá mais ou menos como mostra o gráfico a seguir:

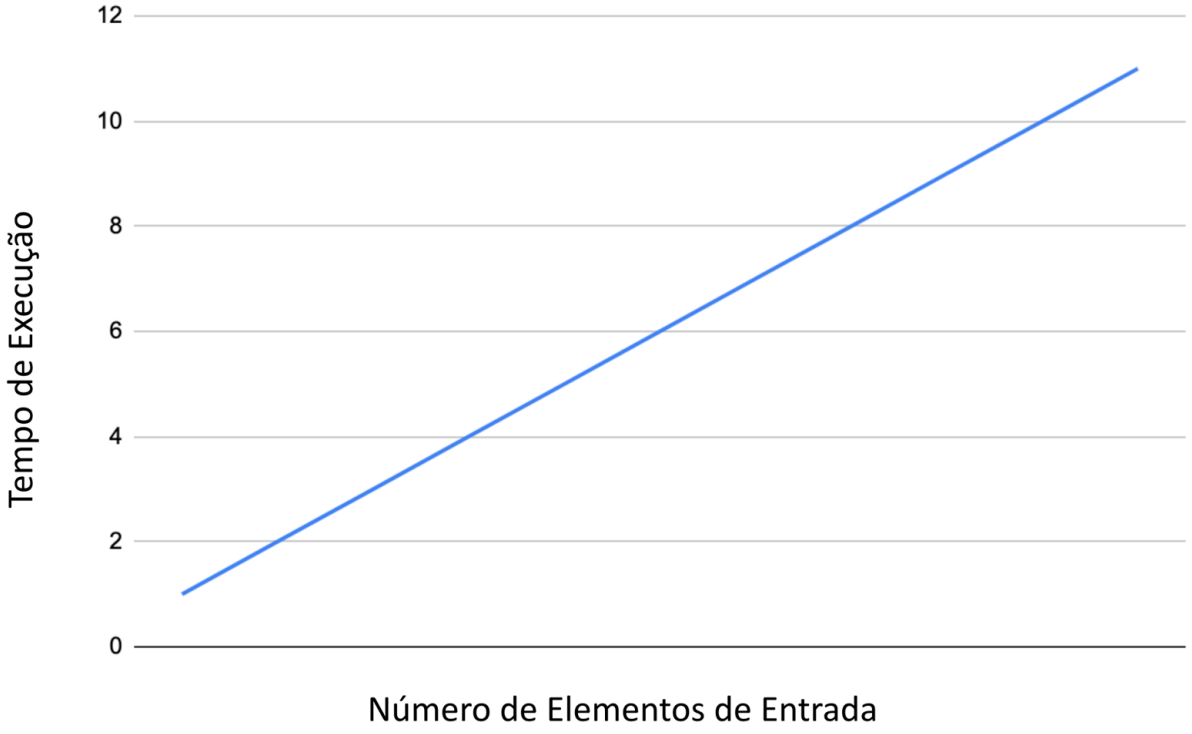


Figura 4.2: Complexidade linear — Tempo de execução x Número de elementos de entrada

Vamos fazer um teste de desempenho comparando as duas soluções. Vamos começar com uma lista com 100 clientes e 100 pagamentos, e multiplicar por 10, até chegar a 100 mil clientes e pagamentos. Para cada teste, vamos executar 100 requisições e verificar a média do tempo de resposta para cada método utilizado. A figura a seguir mostra o resultado desse experimento:

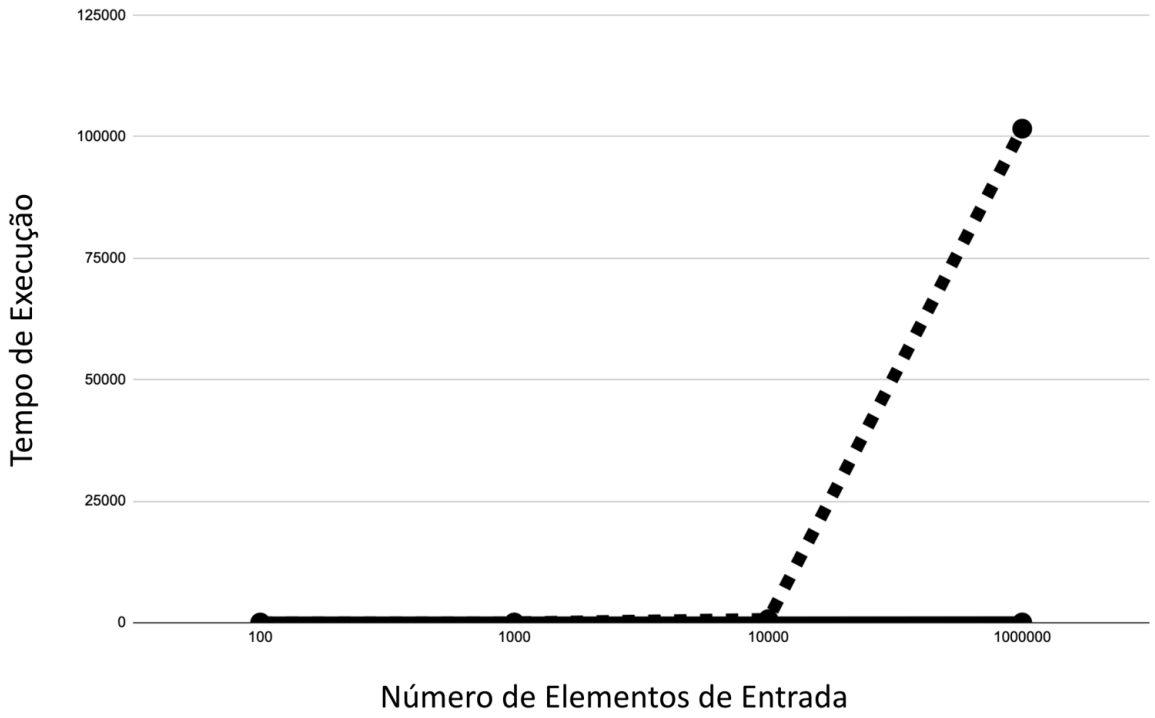


Figura 4.3: Tempo de execução x Número de clientes e pagamentos

Como é possível verificar na imagem, na primeira implementação (linha pontilhada), que tem uma complexidade quadrática, o tempo de execução começa em 1 milissegundo nas listas com 100 elementos, mas acaba com mais de 100 mil milissegundos (100 segundos!). Por outro lado, o crescimento do tempo de execução com o segundo método (linha normal) é muito menor: de 1 milissegundo com as listas de 100 elementos, passa para apenas 27 milissegundos nas listas com 100 mil elementos.

A diferença é tão grande que fica até difícil de visualizar no gráfico, por isso veja na tabela a seguir a diferença no tempo de execução em todos os casos de teste:

Elementos	N	N^2
100	1	1
1000	1	8
10000	3	711
100000	27	101670

Com esse exemplo simples, é possível verificar a diferença enorme que existe nas duas implementações. Verificar a complexidade da implementação é essencial, principalmente em métodos que são muito utilizados ou que processam uma grande quantidade de dados, pois uma pequena melhoria nos algoritmos pode otimizar muito tanto o desempenho quanto a escalabilidade da aplicação. Os algoritmos (e suas complexidades) também são bastante acompanhados de estruturas de dados como listas, *hashes* e árvores, por isso é também bastante importante que os desenvolvedores conheçam bem essas estruturas.

4.2 Cache

Além da implementação dos algoritmos, outra característica que pode deixar um trecho de código lento é o acesso a recursos externos. Existem diversos tipos de recursos que podem ser acessados, por exemplo:

- **Chamadas de rede:** um sistema pode ter que consultar informações em outras aplicações, como no exemplo do PIX, em que pode ser necessário acessar outra aplicação que contenha informações sobre possíveis chaves de pagamento que são suspeitas de fraudes;

- **Acesso a arquivos:** muitas aplicações precisam acessar arquivos como imagens, vídeos ou arquivos binários. Esses acessos podem ser lentos dependendo do tipo do hardware e do tamanho do arquivo;
- **Acesso a banco de dados:** praticamente todas as aplicações armazenam dados em um banco de dados, que pode ser relacional ou NoSQL. Em muitos casos o banco de dados está em um servidor diferente do da aplicação, o que pode tornar o acesso lento ou deixar a execução de uma consulta pesada e demorada.

Outra coisa importante é que o acesso a recursos tem diversos riscos: uma chamada de rede pode ser feita para um recurso que não existe mais, ou esse recurso pode estar tendo dificuldades que vão fazer essa chamada ter um *timeout*, por exemplo.

Por isso, em diversos casos, podemos utilizar um **cache**, que é uma estrutura normalmente em memória que armazena temporariamente dados para evitar que a chamada tenha que ser feita, ou fazer a resposta a essa chamada ser muito mais rápida do que seria se o processamento completo fosse feito. O cache pode ser implementado de várias formas; a forma mais simples é utilizar uma estrutura de dados como o `HashMap` no Java.

No exemplo a seguir, temos duas rotas que fazem a mesma coisa: *buscam uma pessoa em um banco de dados*. A diferença é que a segunda implementação possui um cache simples, e antes de verificar se a pessoa existe no banco de dados, é verificado primeiro no cache; se essa pessoa existir no cache, é retornada a versão do cache; se não existir, é feita a busca no banco de dados.

```
HashMap<Integer, Pessoa> cache = new HashMap<>();

@GetMapping("/pessoa/{id}")
public Pessoa getPessoa(@PathVariable Integer id){
    return pessoaRepository
            .findById(id)
```

```
            .orElseThrow(() -> new RuntimeException());
}

@GetMapping("/pessoa/{id}/cache")
public Pessoa getPessoaCache(@PathVariable Integer id){
    return pessoaCache(id);
}

public Pessoa pessoaCache(Integer id) {
    if (cache.containsKey(id)) {
        return cache.get(id);
    } else {
        Pessoa  pessoa = pessoaRepository
                .findById(id)
                .orElseThrow(() -> new RuntimeException());
        cache.put(pessoa.getId(), pessoa);
        return pessoa;
    }
}
```

Vamos testar agora as duas implementações para verificar o tempo de execução das rotas. O teste foi feito com o banco de dados PostgreSQL (também conhecido como Postgres), mas o resultado será parecido em qualquer banco de dados.

Na execução da rota sem o cache, o tempo médio de execução para 100 requisições foi de 15 milissegundos; já com o cache, o tempo médio ficou em 1 milissegundo. A diferença é grande, mas poderia ser ainda maior caso a consulta executada fosse mais complexa ou caso o banco de dados estivesse executando em uma máquina diferente da aplicação, pois nesse caso ainda teria o tempo de transmissão dos dados na rede.

Apesar de ser possível implementar um cache assim, isso não é muito recomendado. Existem diversos problemas que temos que tratar quando implementamos um cache, como acessos concorrentes e políticas de remoção de chaves. Por isso, é sempre recomendável utilizar uma *ferramenta de cache*. Alguns exemplos

existentes no mercado são o Redis e o HazelCast. Diversos frameworks, como o Spring, já possuem implementações prontas para facilitar o uso de cache.

Temos que tomar cuidado para que os dados no cache sejam sempre removidos ou renovados quando necessário. Ainda seguindo nosso exemplo, quando uma pessoa é excluída do banco de dados, também devemos excluí-la do cache, assim como quando os dados de uma pessoa são atualizados no banco de dados, eles também precisam ser atualizados no cache.

4.3 Índices

Praticamente toda aplicação acessa um banco de dados, seja ele relacional ou não relacional. Em uma aplicação que faz consultas em grandes conjuntos de dados, é essencial analisar as consultas que são pesadas e verificar se todas essas consultas usam um índice na sua execução; se não usarem, provavelmente as consultas não estão otimizadas e estão levando mais tempo do que poderiam para executar.

Normalmente, qualquer banco de dados cria um índice para as colunas que são chaves – por exemplo, as colunas que são o `id` de uma tabela ou de uma coleção, ou colunas que são marcadas com alguma restrição como o `UNIQUE`. Porém, para as outras colunas, os índices devem ser criados explicitamente.

Por exemplo, imagine uma tabela que tem os dados de uma pessoa em um banco de dados relacional com as colunas *nome*, *CPF* e *data de nascimento*. Normalmente, em um banco de dados relacional, essa tabela possui um `id` artificial sequencial e que, automaticamente, possuirá um índice. Porém, outros campos poderão ser bastante utilizados para a busca de uma pessoa, como

o CPF ou a data de nascimento, caso a aplicação que usa esse banco possua filtros pela idade das pessoas.

O script a seguir mostra como pode ser feita a criação dessa tabela:

```
CREATE TABLE pessoa
(
    id bigint NOT NULL,
    nome character varying NOT NULL,
    cpf character varying NOT NULL,
    data_nascimento date NOT NULL,
    CONSTRAINT pessoa_pkey PRIMARY KEY (id)
)
```

Para fazer alguns testes de desempenho, foram inseridos 1 milhão de registros nessa tabela em um banco de dados Postgres. Vamos comparar algumas consultas simples para verificar a diferença de desempenho entre elas. As consultas a seguir são bastante parecidas, uma faz a busca pelo `id` de uma pessoa e a outra faz a busca pelo `cpf` da pessoa.

```
SELECT id, cpf, data_nascimento, nome
FROM public.pessoa
WHERE id = 900001

SELECT id, cpf, data_nascimento, nome
FROM public.pessoa
WHERE cpf = '900000'
```

Apesar de os códigos serem praticamente iguais, tem uma diferença importante no desempenho das duas consultas: enquanto a primeira tem uma média de tempo de execução de 60 milissegundos, a segunda tem uma média de 93 milissegundos – mais de 50% a mais do tempo da primeira consulta.

Isso acontece porque a primeira consulta usa um índice, enquanto a segunda faz uma operação chamada de *full table scan*, isto é, ela faz uma varredura na tabela inteira até encontrar o valor desejado. É

possível verificar essa diferença utilizando uma ferramenta de análise de consultas.

As imagens a seguir mostram que a primeira consulta usou um índice, enquanto a segunda fez a varredura na tabela.

Figura 4.4: Consulta usando o índice da chave primária

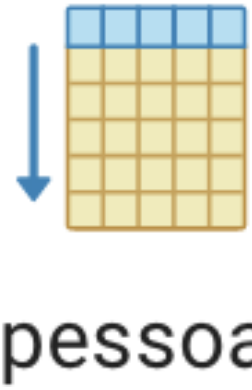


Figura 4.5: Consulta que faz um full table scan

Podemos resolver o problema de a segunda consulta não estar otimizada criando um índice na coluna `cpf` com o seguinte script SQL:

```
CREATE INDEX cpf_index ON public.pessoa (cpf);
```

Com isso, o tempo de execução da consulta de uma pessoa pelo CPF também fica próximo dos 60 milissegundos. A criação de índice é uma tarefa bastante importante, e existem ferramentas de análise

das consultas que são realizadas que podem indicar a falta de um índice em uma determinada tabela.

Isso vale também para alguns bancos de dados não relacionais, como o MongoDB. Por exemplo, o serviço Atlas, que disponibiliza o MongoDB como um serviço na nuvem, possui um analisador de consultas que indica consultas que poderiam se beneficiar da criação de índices. Para saber mais sobre o serviço, acesse: https://www.mongodb.com/docs/atlas/tutorial/profile-database/.

4.4 Comunicação assíncrona

Quando implementamos o back-end de uma funcionalidade, normalmente o primeiro pensamento é fazer isso de forma síncrona. Isso quer dizer que, quando um serviço é chamado, ele já retorna o seu resultado final, seja sucesso ou falha. Por exemplo, na finalização de uma compra, normalmente é chamada uma rota que fecha a compra e envia as informações da compra, do pagamento e da entrega. Para confirmar a compra, será necessário primeiro verificar se todos os produtos estão realmente disponíveis, depois analisar o endereço de entrega e calcular o preço do frete e, por fim, enviar os dados do pagamento para autorização em um serviço, provavelmente um *gateway* de pagamento.

Esse processo todo pode ser demorado. Imagine em uma época como o Natal ou a Black Friday, quantos pagamentos estão sendo realizados? Pode ser que o *gateway* de pagamento simplesmente não funcione, ou, mesmo que funcione, demore alguns longos segundos para responder. Por isso, fazer esse processamento de forma síncrona, isto é, tentar processar a compra na mesma rota que recebe o pedido, pode ser um alto risco para a empresa, já que, se algum dos pontos falhar, a compra de um cliente pode ser perdida.

Uma forma de resolver esse problema é fazer esse processamento de forma *assíncrona*. Nesse modelo, existirá um processo que receberá os dados e outro que fará o processamento. O processamento do dado não precisa ser necessariamente feito assim que os dados são recebidos. O processo que recebe os dados pode até fazer alguma validação simples, como verificar se todos os dados obrigatórios foram enviados, mas a principal responsabilidade dele será colocar o dado em uma *fila*. Depois, um outro processo vai ficar analisando a fila; caso ela esteja vazia, o processo não faz nada, mas, se algum dado chegar, o processo recebe os dados que estão na fila e os processa.

Com esse mecanismo, o processo que recebe os dados será extremamente simples e rápido, então, mesmo que haja um grande crescimento no número de requisições, o processo suportará a carga, o que aumentará bastante a escalabilidade da aplicação. Já na outra parte da aplicação, o processamento dos dados vai ser executado conforme a fila vai sendo preenchida; se muitos dados chegarem à fila, acima da capacidade de processamento, não há problema: o dado ficará esperando até que ele seja processado.

Pensando em um exemplo de um e-commerce, podemos ter um serviço que apenas recebe o pedido, mas ele não vai tentar processar o pedido, e sim apenas colocá-lo em uma fila. Depois de algum tempo (que pode até ser instantâneo, se a fila estiver vazia), outro processo pegará os dados desse pedido e tentará fazer todos os processos lentos: validar o estoque, calcular o preço do frete e autorizar o pagamento via cartão de crédito. Nessa implementação, normalmente os pedidos terão um *estado*. Por exemplo, quando o pedido é recebido, ele fica no estado *AGUARDANDO*; depois, quando ele começa a ser processado, ele pode passar para o estado *PROCESSANDO* e, por fim, ele pode ter um estado *SUCESSO*, caso tudo dê certo, ou *ERRO*, se acontecer algum problema nas verificações do pedido.

A grande maioria dos e-commerces é implementada assim atualmente, e é fácil perceber isso, pois quando fazemos um pedido

atualmente, primeiro nos é enviado apenas um e-mail confirmando que a compra foi realizada. Depois de algum tempo, é enviado outro e-mail informando se o pagamento foi processado com sucesso ou não. Esse tempo entre o pedido e a confirmação do pagamento pode variar: se for em um período em que poucas compras são feitas, provavelmente ele será bem rápido, pois a fila de pedidos estará vazia ou com poucos pedidos; em épocas em que há grande demanda, a confirmação pode demorar algumas horas, pois a fila pode estar grande.

Esse tipo de processamento é muito utilizado em casos onde há grande variação de demanda, e é difícil garantir a escalabilidade caso o número de requisições aumente, ou em casos onde o processamento é lento e pode demorar mais de alguns poucos segundos. Existem diversas ferramentas que podem ajudar bastante no gerenciamento de filas; duas bastante conhecidas são o Apache Kafka e o RabbitMQ.

Conclusões

Obviamente existem diversas outras melhorias que podem ser feitas no código para aumentar o desempenho e a escalabilidade de uma aplicação. Neste capítulo, vimos alguns exemplos de erros comuns que pessoas programadoras menos experientes cometem e que afetam o desempenho de suas aplicações. Por isso, antes de simplesmente melhorar a qualidade do hardware ou tentar implementar uma arquitetura supercomplexa em sua aplicação, faça análises no seu código para tentar descobrir possíveis gargalos e verificar se ele pode ser melhorado, seja com melhores algoritmos, com a utilização de cache ou qualquer outra técnica que possa melhorar o seu desempenho.

A melhoria no código da aplicação muitas vezes pode ser o suficiente para melhorar o seu desempenho, porém existem casos em que não há mais o que melhorar no código e ainda assim a

aplicação apresentar lentidão. Por isso, essa melhora terá que ser alcançada via melhorias nos recursos de hardware que executam a aplicação. Veremos algumas dessas possíveis melhorias no próximo capítulo.

Capítulo 5
Desempenho e escalabilidade na infraestrutura

Vimos no capítulo anterior que é possível melhorar o desempenho e a escalabilidade de uma aplicação fazendo várias melhorias no código. No entanto, há um limite para essas otimizações. Dependendo da quantidade de requisições que uma aplicação precisa atender ou da quantidade de dados que ela precisa processar, mesmo com os melhores algoritmos, a infraestrutura também vai virar um gargalo. Por isso, em algum momento será necessário aumentar os recursos computacionais para executar a aplicação de forma adequada.

Existem diversas melhorias que podem ser implementadas na infraestrutura de uma aplicação para aprimorar o seu desempenho. A solução mais óbvia é simplesmente utilizar hardware de melhor qualidade. Por exemplo, uma máquina com 16 processadores provavelmente conseguirá atender a um maior número de requisições em comparação a uma máquina com apenas 4 processadores. Da mesma forma, uma máquina com 128 GB de memória RAM conseguirá processar uma quantidade significativamente maior de dados do que uma com 16 GB.

No entanto, nem sempre a solução para melhorar o desempenho e a escalabilidade de uma aplicação é simplesmente adquirir uma máquina de maior capacidade, já que o custo envolvido pode ser exorbitante. Portanto, é necessário explorar outras alternativas de melhoria da infraestrutura para alcançar esses objetivos. A seguir, analisaremos algumas opções viáveis para otimizar o desempenho e a escalabilidade de nossas aplicações.

5.1 Melhoria de hardware

Como dito anteriormente, às vezes chegamos ao limite do que uma aplicação pode atender em um hardware específico. Por isso, temos que melhorar os recursos que nossas aplicações podem utilizar. Por exemplo, é certo que aumentar o número de CPUs do servidor que executa a aplicação fará aumentar também o número de requisições que a aplicação consegue atender.

Em uma aplicação web, normalmente o software do servidor ou do contêiner que executa essa aplicação (como o Tomcat, JBoss ou Apache) cria um *pool* de *threads* que atende às requisições de usuários. Se todas as *threads* estão ocupadas atendendo requisições, um usuário terá que esperar até ser atendido. Com um número de CPUs maior, as *threads* concorrerão menos por CPU, o que fará as requisições terem mais tempo de CPU e um tempo de resposta melhor para cada uma. Além disso, essas *threads* também ocupam memória, por isso é importante também ter uma quantidade de memória suficiente para que mais *threads* possam ser criadas.

Por exemplo, o citado Tomcat, que é um dos contêineres de aplicações web mais utilizados para a plataforma Java, possui um tipo de *thread* chamada de ***worker***, que processa as requisições de um usuário. Normalmente, quando ela é iniciada, ela já processa 10 *threads* que podem atender requisições, indo até um máximo de 200 por padrão (esse número é configurável). Em uma máquina que tem apenas uma CPU, essas 10 *threads* vão disputar a única CPU disponível. Se o número de requisições for pequeno, o desempenho será aceitável, porque normalmente algumas dessas *threads* estarão paradas. No entanto, se muitas requisições simultâneas forem feitas, todas as *threads* tentarão usar a CPU ao mesmo tempo, o que pode deixar a aplicação lenta. Se melhoramos os recursos que a aplicação está usando para, por exemplo, 4 CPUs, as *threads* terão muito mais recursos para processar requisições em paralelo, o que melhorará

bastante o desempenho da aplicação no caso de múltiplas requisições simultâneas.

Para deixar essa melhora mais palpável, com números reais, vamos fazer alguns testes usando máquinas no servidor de nuvem AWS da Amazon (os mesmos testes poderiam ser feitos em outros provedores de nuvem, como o GCP ou Azure). O AWS possui um serviço chamado **EC2** (*Elastic Compute Cloud*), que disponibiliza máquinas virtuais com diversos números de recursos. Nele, podemos criar instâncias que serão cobradas por hora de utilização.

A tabela a seguir mostra alguns exemplos das configurações possíveis de serem utilizadas:

Nome	CPU	Memória (GB)
t2.nano	1	0,5
t2.micro	1	1
t2.small	1	2
t2.medium	2	4
t2.large	2	8
t2.xlarge	4	16
t2.2xlarge	8	32

Para comparar como os recursos podem influenciar o desempenho da aplicação, vamos criar duas máquinas: uma que possui apenas 1 CPU e 2 GB de memória RAM (*t2.small*) e outra com 2 CPUs e 4 GB de memória (*t2.medium*). Utilizando o JMeter, faremos uma grande quantidade de requisições. Para testar o desempenho das máquinas, vamos utilizar uma aplicação que possui apenas a rota que busca os pagamentos dos clientes, com o mesmo código desenvolvido para o capítulo anterior.

```
@GetMapping
public List<Cliente>  getValue() {

    Map<String, Integer> clientesPagamentos = new HashMap<>();
    for (Pagamento  pagamento : pagamentos) {
        clientesPagamentos.merge(pagamento.getCpf(),
                pagamento.getValor(), Integer::sum);
    }

    for (Cliente cliente : clientes) {
        Integer valor = clientesPagamentos
            .get(cliente.getCpf());
        cliente.setTotalPagamentos(valor);
    }
    return clientes;
```

```
}
```

No teste, foram feitas requisições simultâneas para ver até quantas requisições os servidores se mantinham com um tempo de resposta aceitável. Na máquina *t2.small*, a aplicação funcionou com 5 e 50 requisições simultâneas, e o tempo de resposta apenas dobrou com um número 10 vezes maior de requisições. Porém, com 100 requisições simultâneas, o servidor não aguentou a quantidade de requisições, e 100% delas falharam.

Já com a *t2.medium*, o tempo de resposta para 5 requisições simultâneas é bem próximo do da *t2.small*, mas acima desse número o desempenho é muito melhor em comparação ao primeiro exemplo, principalmente com 100 requisições simultâneas. Mesmo com essa grande quantidade de requisições, o servidor continuou funcionando bem e com um bom tempo de resposta. Com 200 requisições simultâneas, a *t2.medium* também começa a ter um desempenho ruim, a resposta começa a demorar quase 1 minuto. A tabela a seguir compara o desempenho dessas duas máquinas:

Requisições simultâneas	**t2.small**	**t2.medium**
5	1.4s	1.4s
50	2.8s	1.9s
100	-	3s
200	-	58s

A principal explicação para o aumento do tempo de resposta é a quantidade de *threads* que podem tratar as execuções. Como as máquinas possuem um número limitado de CPUs, a quantidade máxima de *threads* que conseguem fazer processamento paralelo é limitada. Já nos casos em que houve erro na execução, a explicação é que a aplicação estourou a quantidade de memória que a máquina suportava, por isso as requisições não puderam ser processadas.

5.2 Balanceamento de carga

Melhorar os recursos computacionais é fácil, mas pode ser caro. Os custos dos servidores nem sempre são proporcionais aos recursos disponibilizados, isto é, uma máquina que tem o dobro de capacidade pode custar muito mais que o dobro. Uma solução que pode ter efeitos parecidos com os da melhoria de hardware, mas que pode ter um custo menor, é ter mais de uma instância da aplicação e utilizar um **balanceador de carga** para dividir as requisições entre as diferentes instâncias das aplicações.

Existem diversas ferramentas que fazem balanceamento de carga. A AWS possui o LB (*Load Balancer*; em português, balanceador de carga), que funciona diretamente com as instâncias do EC2. A criação de um LB na AWS é bastante simples, precisamos apenas configurar as portas para as quais queremos fazer o redirecionamento para as máquinas. Neste caso, em vez de enviar as requisições diretamente para o servidor, precisamos fazer as requisições para o LB, e ele fará o redirecionamento para uma ou mais máquinas do EC2.

Vamos testar a mesma aplicação agora, mas, em vez de melhorar os recursos das instâncias, vamos criar duas instâncias iguais e configurar um LB para distribuir a carga entre as duas:

Requisições simultâneas	t2.small	t2.medium	2x t2.small
5	1.4s	1.4s	1.6s
50	2.8s	1.9s	3.2s
100	-	3s	4.6s
200	-	58s	41s - 35% de erro

Vemos nos resultados que o tempo de resposta para o teste com 200 requisições simultâneas ficou até melhor com o LB, mas isso aconteceu principalmente porque a taxa de erro ficou alta (cerca de 35%), o que mostra que o desempenho de 2 instâncias do *t2.small* é pior do que o de uma única instância do *t2.medium*. Isso acontece principalmente por causa da memória: enquanto uma *t2.medium* tem 4 GB de memória, a *t2.small* tem 2 GB — com duas instâncias, tínhamos apenas 1 GB de memória para cada. Podemos comprovar isso pelo erro que começou a aparecer no log da aplicação quando a carga ficou muito alta, que foi um `java.lang.OutOfMemoryError`.

Além da melhora de desempenho mostrada nos testes anteriores, o Load Balancer é ainda uma excelente estratégia para melhorar a *confiabilidade* da aplicação, já que teremos duas instâncias da aplicação sendo executadas, mas isso é assunto para os próximos capítulos.

5.3 Cache na infraestrutura

No capítulo anterior, discutimos como implementar um cache diretamente no código. Porém, é possível também fazer caches diretamente na infraestrutura, sem que seja necessário fazer alterações no código.

Esse tipo de cache serve principalmente para arquivos estáticos, como arquivos HTML, imagens e vídeos. Na maioria dos casos, esses tipos de arquivos são sempre iguais, independente dos usuários que estão acessando o conteúdo, por isso é bem mais fácil manter cópias desses arquivos em vários pontos da internet.

Uma das principais formas de implementar esse tipo de cache é utilizar uma CDN (*Content Delivery Network*, ou Rede de Distribuição de Conteúdo, em português). Nesse tipo de rede, existem diversos servidores distribuídos geograficamente que armazenam cópias dos conteúdos.

Essa rede oferece três benefícios principais, que são:

1. **Melhor desempenho:** a distância física é bastante importante na velocidade de distribuição de um conteúdo. Se um conteúdo for distribuído no Brasil inteiro, por exemplo, e um usuário do Amazonas precisar acessar esse conteúdo que está em um servidor em São Paulo, esse acesso será bem mais lento do que se esse conteúdo estivesse em um servidor no próprio estado do Amazonas;

2. **Escalabilidade:** se todos os usuários de uma aplicação acessam um servidor central, a chance de esse servidor não suportar a carga de distribuição desse conteúdo é grande. Porém, se a distribuição é feita por vários servidores espalhados geograficamente, a chance de ocorrer um problema de escalabilidade é bem menor;

3. **Tolerância a falhas:** esse é um assunto para os próximos capítulos, mas uma CDN também aumenta a confiabilidade da aplicação. Se um servidor que possui um conteúdo para de funcionar, não há maiores problemas, pois existem outros servidores redundantes que podem distribuir o conteúdo.

Obviamente existem problemas também na utilização desse cache, como o alto custo de manter uma rede de servidores distribuída geograficamente e a atualização de conteúdo, que pode demorar

alguns segundos ou até alguns dias. Esse tipo de cache é muito utilizado por empresas que precisam distribuir grandes quantidades de conteúdos estáticos, como as redes sociais, que permitem o compartilhamento de mídias como fotos e vídeos, e empresas de *streaming* de vídeo. Grandes empresas muitas vezes possuem suas próprias CDNs, mas empresas menores podem usar serviços como o CloudFare e o AWS CloudFront para a criação desse tipo de rede.

Conclusões

Os testes de desempenho feitos com máquinas com diferentes recursos disponíveis mostram que uma pequena melhoria no hardware já melhora bastante o desempenho da aplicação. Por isso, é bastante importante saber dimensionar bem os recursos que serão utilizados pela aplicação, não só para que ela não fique lenta pela falta deles, mas também para que eles não sejam desperdiçados, o que aumentaria o custo de execução da aplicação.

Nesta parte do livro, vimos que, para alcançar um desempenho e uma escalabilidade adequados, precisamos nos preocupar tanto com o código quanto com a infraestrutura das aplicações. Na Parte 3 deste livro, veremos que a mesma coisa acontece com a *confiabilidade* da aplicação, pois esse requisito também vai depender de um código que seja escrito pensando em possíveis erros e de uma infraestrutura que tenha monitoramento e redundância para evitar que um problema de infraestrutura faça a aplicação parar de funcionar.

Parte 3: Confiabilidade

Capítulo 6
Confiabilidade

Toda aplicação que tenha um número grande de usuários terá problemas. Mesmo que ela seja muito bem implementada, sempre haverá falhas, seja no código da aplicação ou na sua infraestrutura. Por isso, assim como vimos nos capítulos sobre desempenho, temos que utilizar estratégias para aumentar a confiabilidade dos sistemas tanto no código quanto na infraestrutura, para que, quando uma falha ocorrer, ela cause o mínimo de problemas possível na aplicação e esta consiga se recuperar o mais rápido possível.

Além de erros no código ou na infraestrutura, alguns erros podem ser causados pelo próprio usuário da aplicação. Por exemplo, podemos ter um campo que espera um número inteiro, mas o usuário tenta enviar um texto. Devemos adicionar o máximo de validações possíveis para evitar que usuários façam alguma coisa errada, mas há casos em que não há como evitar esse tipo de situação. Além das validações, temos que conseguir reconhecer e tratar esses erros para que eles sejam comunicados corretamente para os usuários quando ocorrerem. Neste capítulo, veremos alguns dos principais requisitos não funcionais que são ligados à confiabilidade das aplicações, como a *tolerância a falhas* e a *disponibilidade*.

6.1 Tolerância a falhas

Como dito no início deste capítulo, erros acontecerão em toda aplicação que tenha um número razoável de usuários. Alguns serão problemas de infraestrutura, como quando um servidor de uma

aplicação fica lento porque o espaço em disco acabou, ou porque o banco de dados começou a gerar um monte de logs, pois a equipe de BI, de um dia para o outro, começou a gerar muito mais relatórios do que o esperado (caso real acontecido comigo alguns anos atrás). Outros problemas estarão no código da aplicação; por exemplo, em um campo que é obrigatório em uma tabela no banco de dados, mas que a pessoa que programou se esqueceu de fazer a validação na entrada do usuário, e houve uma tentativa de salvar um registro no banco de dados sem esse campo. Isso gerará uma exceção no banco de dados com um monte de mensagem que será ilegível para um usuário da aplicação. O erro pode ser causado até mesmo pelo usuário: você avisa à pessoa que uma senha deve ter entre 8 e 10 caracteres, com pelo menos um caractere especial, mas ela insiste em tentar usar a senha 123456.

Veja que é praticamente impossível implementar uma aplicação 100% à prova de falhas. O que devemos fazer nesse caso é tratar os erros da melhor forma possível, tentando, sempre que uma falha acontecer, remediá-la e dar uma resposta satisfatória ao usuário. Isso é chamado de **tolerância a falhas**. Por exemplo, em um caso de erro interno, como a queda de um serviço, devemos ter uma tela que informe corretamente para os usuários que o sistema está passando por dificuldade, e, se possível, com alguma informação sobre o que está sendo feito e uma previsão de quando o sistema voltará a funcionar.

Para erros no código, como a tentativa de inserção de dados inválidos em uma tabela do banco de dados, deve-se usar o *tratamento de exceção*, disponível em praticamente qualquer linguagem de programação atual. Além disso, o erro deve ser tratado para retornar uma informação útil para o usuário. Um erro comum de várias aplicações é simplesmente retornar a mensagem gerada no servidor, o que muitas vezes contém informações técnicas e que não interessam ao usuário.

Em APIs REST, outro ponto importante é o código de resposta de um serviço. Esses códigos possuem algumas faixas de resposta, e as

principais são:

- As **2xx**, que indicam que o serviço foi executado corretamente, como `200 OK`, `201 CREATED` e `204 NO_CONTENT`;
- As **4xx**, que indicam algum erro do usuário, como `400 BAD_REQUEST`, `401 UNAUTHORIZED` e `404 NOT_FOUND`;
- As **5xx**, que indicam um erro do servidor, como `500 INTERNAL_SERVER_ERROR`.

Retornar erros que fazem sentido em rotas REST é bastante importante, pois, com essa informação, um cliente pode decidir o que fazer. Por exemplo, se a resposta for `404`, não adianta tentar chamar o serviço novamente, pois o recurso não existe no servidor. Já se o erro foi um `401` (não autorizado), a pessoa sabe que não está autenticada, então ela tentará fazer a autenticação no sistema, ou desistirá, porque sabe que não tem permissão para acessar aquele recurso. No próximo capítulo, veremos algumas estratégias para a validação de erros e como retorná-los corretamente para os usuários.

6.2 Disponibilidade

Um dos principais indicadores de confiabilidade de uma aplicação é a sua **disponibilidade**, isto é, do total de tempo possível, o quanto a aplicação estava operando normalmente. Por exemplo, em um período de 24 horas, se a aplicação esteve disponível por apenas 12 horas, a disponibilidade dela foi de apenas 50% – obviamente, 50% é um número inaceitável para praticamente qualquer tipo de aplicação comercial.

Grande parte das aplicações tem o objetivo de ter uma disponibilidade de 100%, e isso é possível de obter em alguns períodos de tempo. Grandes aplicações ficam disponíveis 100% do tempo durante dias, talvez até meses. Porém, mesmo as melhores e

mais importantes aplicações têm problemas de vez em quando. Um dos tipos de aplicações mais importantes que existem são os meios de pagamentos, mas nem mesmo eles possuem disponibilidade em 100% do tempo - você já deve ter passado por essa experiência em algum momento.

A disponibilidade de uma aplicação é tão importante que diversos serviços definem o mínimo de disponibilidade que ele deve ter em um período de tempo. Por exemplo, a AWS define que, se a disponibilidade ficar abaixo de 99% durante um mês, ela devolve parte do dinheiro pago pelo cliente naquele mês como crédito para as próximas faturas. Outros serviços, como o GitHub, possuem páginas que monitoram todos os serviços da plataforma em tempo real e mostram a disponibilidade desses serviços.

A figura a seguir mostra um exemplo dessa página:

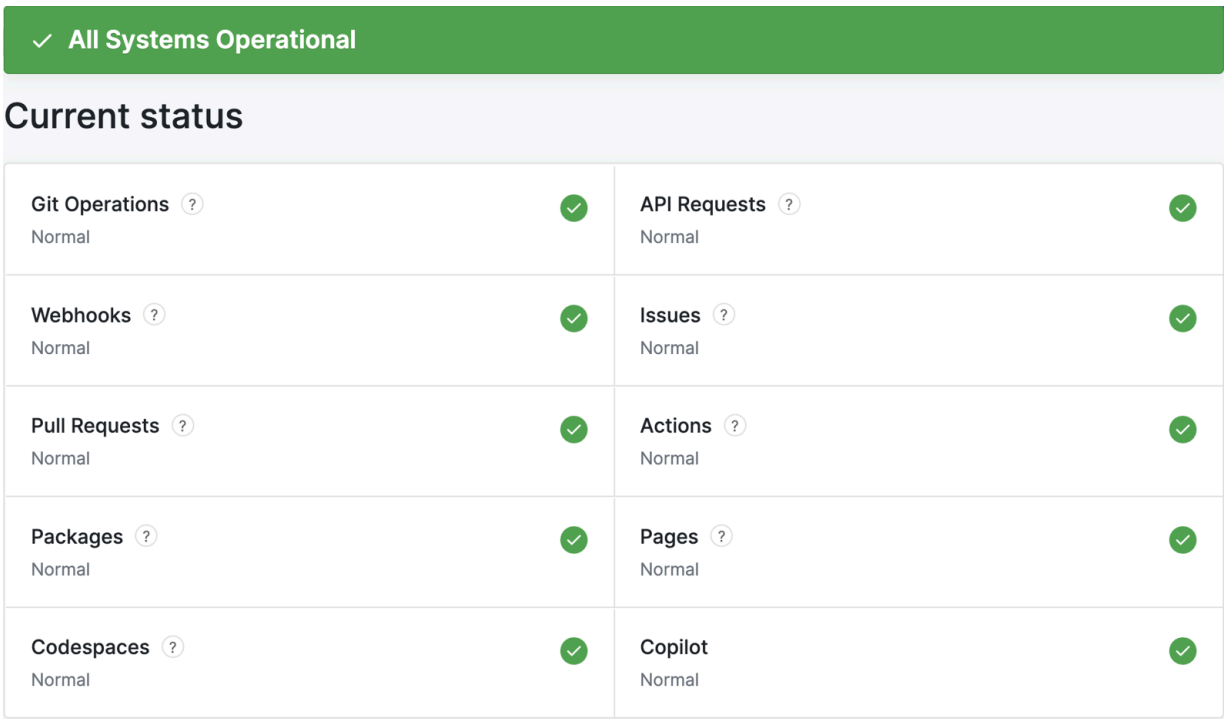


Figura 6.1: Disponibilidade dos serviços no GitHub

Mas como manter a disponibilidade da aplicação em um nível alto? Esse é um grande desafio. Uma das técnicas mais usadas já foi até

discutida no capítulo anterior, que é o balanceamento de carga. Se uma aplicação é executada em apenas um servidor e este servidor cair, a aplicação inteira para de funcionar. Por isso, fazer o balanceamento de carga em dois ou mais servidores, executando instâncias separadas da aplicação, aumenta bastante a confiabilidade e a disponibilidade da aplicação.

Mas fazer apenas o balanceamento de carga pode não ser o suficiente para manter a disponibilidade em um nível altíssimo. Por exemplo, se uma aplicação está sendo executada em um *data center* próprio de uma empresa, e algum problema de rede acontecer nesse *data center*, a aplicação ficará indisponível do mesmo jeito. Por isso, aplicações muito críticas terão diversas instâncias sendo executadas em diversos *data centers* distribuídos geograficamente.

Em um provedor de nuvem, temos exatamente o mesmo problema: se a aplicação está sendo executada em apenas uma área geográfica de um provedor de nuvem e essa área desse provedor ficar indisponível, a aplicação ficará indisponível também, mesmo que existam várias máquinas rodando instâncias diferentes da aplicação. Uma possível solução para isso é executar a aplicação em diversas áreas geográficas da nuvem, o que todos os grandes provedores de nuvem possibilitam de maneira relativamente fácil.

Mesmo rodando diversas instâncias em áreas geográficas diferentes, ainda existe uma possibilidade, mesmo que pequena, de que sua aplicação fique indisponível se houver um grande problema no provedor de nuvem. Por isso, existem ainda aplicações que são executadas em diferentes provedores de nuvem, o que não é tão simples, mas é possível de implementar. Obviamente, isso é feito apenas para aplicações extremamente críticas, e o custo pode ser bastante alto.

6.3 Monitoramento

Ter disponibilidade é importante, mas um dia a aplicação pode ficar indisponível no meio da madrugada, e como vamos saber que isso aconteceu? É extremamente importante fazer o **monitoramento** de todos os serviços essenciais da aplicação. Podemos monitorar diversos componentes da aplicação: por exemplo, para monitorar o servidor da aplicação, podemos ficar fazendo chamadas de rede em intervalos de tempo, e se por algum motivo uma chamada não funcionar, podemos alertar a equipe de plantão de que algo estranho aconteceu.

O monitoramento também é importante para a coleta de métricas sobre a aplicação. Imagine que, logo que uma aplicação é disponibilizada no mercado, tudo funciona perfeitamente, mas, depois de um tempo, ela tem um crescimento muito grande no número de usuários e começa a ficar cada vez mais lenta. Como vamos descobrir isso? Uma possibilidade é esperar dar algum erro e um cliente abrir um chamado reclamando de lentidão na aplicação, mas esse com certeza não é o melhor cenário.

O ideal é fazer o monitoramento ativo e coletar indicadores sobre a saúde da aplicação, como o tempo de resposta para as requisições do usuário, a quantidade de recursos computacionais que estão sendo usados, o tempo gasto nas consultas ao banco de dados, a quantidade de erros que estão acontecendo na aplicação, entre diversas outras métricas que podem ser coletadas. Essas métricas podem indicar onde estão os possíveis gargalos na aplicação e apontar onde e quando realizar manutenções preventivas, para evitar que os problemas da aplicação apareçam enquanto os usuários a utilizam.

Existem diversas ferramentas de monitoramento: algumas analisam o log da aplicação buscando erros e, caso encontrem algum erro crítico, podem tomar alguma ação, como enviar um e-mail ou uma mensagem via algum comunicador para os responsáveis; outras ficam o tempo todo analisando as métricas da aplicação ou de um

servidor, e se alguma dessas métricas apontarem algum possível problema, ela também pode notificar os operadores de que algum problema está acontecendo. No próximo capítulo, veremos algumas dessas ferramentas e como elas podem ser configuradas em uma aplicação.

6.4 Recuperabilidade

Como já vimos algumas vezes neste capítulo, as aplicações terão falhas em algum momento, mas temos que conseguir nos recuperar o mais rápido possível e sem que esse erro tenha causado nenhum efeito permanente na confiabilidade da aplicação.

Existem diversas estratégias para garantir que a aplicação consiga se recuperar das falhas. Por exemplo, fazer *backups* periódicos do banco de dados de uma aplicação é praticamente uma obrigação. Imagine que um dia o disco do servidor onde estão os dados da aplicação para de funcionar, e não há nenhuma forma de recuperá-lo. Se não existir um backup dele, não há o que fazer. Outro ponto importante é não salvar o backup na mesma máquina onde está o banco de dados, porque, dependendo da falha, ambos serão perdidos. Por isso, ter uma *política de backup* que salva os dados periodicamente em servidores diferentes é o mínimo que toda aplicação deve ter.

É normal um servidor ser reiniciado, mesmo executando em ambientes de nuvem. De tempos em tempos, as máquinas virtuais podem ser reiniciadas por causa de uma manutenção programada do provedor de nuvem. Normalmente essas reinicializações são rápidas, mas se a aplicação não estiver corretamente configurada para se reinicializar sozinha, será necessária uma intervenção humana para reestabelecer os serviços. Por isso, o ideal é que as aplicações usem sempre mecanismos para se recuperarem sozinhas de falhas. No capítulo 8, *Confiabilidade na infraestrutura*, veremos

algumas dessas configurações que podem ser feitas em diferentes níveis, seja no sistema operacional, em um gerenciador de cluster ou até mesmo no gerenciador de contêineres.

Conclusões

Ter uma aplicação confiável é um requisito fundamental. Para o usuário, tentar usar um sistema e, por algum motivo, ele não estar disponível é uma péssima experiência. Obviamente, existem níveis de confiabilidade: alguns sistemas precisam ser praticamente perfeitos e terem disponibilidade próxima a 100% do tempo, e, para isso, vão utilizar vários mecanismos avançados, como multinuvem e servidores que rodam em várias áreas geográficas.

No entanto, essas estratégias possuem custos; por isso, é essencial saber qual o nível de confiabilidade que sua aplicação precisa e quais as estratégias que precisam ser utilizadas para se ter a maior disponibilidade possível. Nos próximos dois capítulos, veremos a implementação de diversas técnicas para atingir os requisitos funcionais discutidos aqui.

Capítulo 7
Confiabilidade no código

Quando ouvimos falar da confiabilidade de uma aplicação, é mais comum pensar na sua infraestrutura. Muitas vezes, quando um sistema para de funcionar, instintivamente falamos que o "servidor caiu". Porém, o código também é muito importante para aumentar a confiabilidade das aplicações. Nos capítulos sobre desempenho e escalabilidade isso já foi demonstrado, pois, se a aplicação não tiver uma boa escalabilidade, é muito provável que ela também não seja confiável, pois um aumento na carga pode diminuir a disponibilidade dela.

Por isso, para melhorar a confiabilidade das aplicações, existem várias técnicas e padrões que podem ser utilizados no código da aplicação, visando evitar ou minimizar problemas para os usuários. Um desses requisitos fundamentais é o tratamento de erros.

É inevitável que ocorram problemas nas aplicações, mas é essencial lidar com eles corretamente e evitar que as falhas causem efeitos adversos no funcionamento geral do sistema; caso contrário, esses problemas podem se agravar ainda mais. Além disso, é importante sempre comunicar aos usuários sobre os problemas que ocorreram na aplicação. Isso permite que eles saibam o que está acontecendo e possam agir de acordo ou, no mínimo, entrar em contato com a equipe responsável.

Também pensando na confiabilidade, devemos instrumentalizar a aplicação para que ela gere indicadores da sua saúde. Por exemplo, coletar dados sobre quais são os métodos que mais estão sendo executados ou os que estão levando mais tempo para executar pode ser importante para detectar possíveis gargalos na aplicação. Portanto, é necessário monitorar ativamente o que está acontecendo internamente na aplicação.

Neste capítulo, vamos explorar algumas técnicas que podemos utilizar diretamente no código da aplicação para aumentar sua confiabilidade.

7.1 Tratamento de exceções

Fazer um tratamento correto das exceções é essencial para evitar que um simples erro cause efeitos indesejáveis na aplicação e diminua a sua confiabilidade. O tratamento de exceções é a forma mais simples de tornar a aplicação **tolerante a falhas**. Qualquer linguagem de programação moderna possui um mecanismo para o tratamento de exceções. Por exemplo, em Java, usamos as palavras reservadas `try`, `catch` e `finally` para isso. No exemplo a seguir, é mostrado um tratamento de exceções para a abertura de um arquivo em Java:

```
public static void main(String[] args) {
    FileReader fileReader = null;
    BufferedReader bufferedReader = null;

    try {
        // Tenta abrir o arquivo
        fileReader = new FileReader("arquivo.txt");
        bufferedReader = new BufferedReader(fileReader);

        //Lê os dados do arquivo
        String linha;
        while ((linha = bufferedReader.readLine()) != null) {
            System.out.println(linha);
        }

    } catch (Exception e) {
        // Se houver algum problema na abertura
        // ou na leitura do arquivo
        System.out.println("Erro na leitura do arquivo: "
            + e.getMessage());
```

```
    } finally {
        // fecha os recursos
        bufferedReader.close();
        fileReader.close();
    }
}
```

Neste trecho de código, a ideia é executar o código que está dentro do bloco `try`. Se nada de errado acontecer, apenas esse bloco será executado. Porém, se algum erro for encontrado e alguma exceção for lançada, será impresso no log da aplicação o erro encontrado. Apesar de o trecho mostrar a ideia básica de como fazer o tratamento de exceções, esse código não é a melhor solução, pois ele faz um tratamento muito genérico das exceções e apenas imprime no log uma mensagem que indica que houve um erro na abertura do arquivo. O ideal é fazer um tratamento mais detalhado dos erros que podem acontecer, como no exemplo a seguir:

```
public static void main(String[] args) {
    FileReader fileReader = null;
    BufferedReader bufferedReader = null;

    try {
        // Tentativa de abrir o arquivo
        fileReader = new FileReader("arquivo.txt");
        bufferedReader = new BufferedReader(fileReader);

        String linha;
        while ((linha = bufferedReader.readLine()) != null) {
            System.out.println(linha);
            // Imprime cada linha do arquivo
        }

    } catch (FileNotFoundException e) {
        // Erro se o arquivo não for encontrado
        System.out.println("Arquivo não encontrado: "
            + e.getMessage());
    } catch (IOException e) {
        // Erro na leitura do arquivo
```

```
            System.out.println("Erro de leitura do arquivo: "
                + e.getMessage());
        } finally {
            try {
                if (bufferedReader != null) {
                    bufferedReader.close(); // Fecha o BufferedReader
                }
                if (fileReader != null) {
                    fileReader.close(); // Fecha o FileReader
                }
            } catch (IOException e) {
                // Pode ocorrer algum erro
                // no fechamento do arquivo também
                System.out.println("Erro ao fechar o arquivo: "
                    + e.getMessage());
            }
        }
}
```

Neste caso, diferenciamos dois possíveis erros, que são o `FileNotFoundException` e o `IOException`. Com eles, é possível retornar para o usuário uma mensagem de erro mais informativa, indicando, por exemplo, que o arquivo não foi encontrado ou que o arquivo existe, mas houve um erro na leitura. Além disso, na hora de fechar a conexão com o arquivo, também podem ocorrer erros, por isso também é importante fazer o tratamento de erro no bloco `finally`.

Uso de recursos

Outro aspecto importante são os recursos que são abertos nas aplicações, como conexões com bancos de dados, acesso a arquivos e criação de novas *threads*. Todos esses recursos ocupam espaço na memória e podem também utilizar tempo de processamento nas aplicações. Por isso, é bastante importante que, quando esses recursos forem criados, lembremos de fechá-los quando o uso deles terminar. Isso pode evitar que ocorra na aplicação o famoso *memory leak*, ou vazamento de memória. Esse tipo de problema pode

diminuir a confiabilidade de uma aplicação, pois um vazamento desse tipo pode fazer a aplicação parar de funcionar.

O código a seguir mostra um exemplo de onde isso pode acontecer. O método deste exemplo abre um arquivo no sistema de arquivos, escreve os dados que estão neste arquivo no console e o fecha ao final. Porém, se algum erro acontecer e ocorrer antes da linha do `reader.close()`, o arquivo não será fechado. Se o programa ou a *thread* onde esse arquivo foi aberto terminar, não haverá maiores problemas, pois o arquivo será fechado automaticamente. Porém, caso o programa ou a *thread* continue sendo executada, o arquivo continuará alocado na memória sem que seja mais utilizado.

```
try {
    BufferedReader reader =
        new BufferedReader(new FileReader("example.txt"));
    String line;

    while ((line = reader.readLine()) != null) {
        // Processa a linha do arquivo
        System.out.println(line);
    }
    reader.close()
} catch (IOException e) {
    e.printStackTrace();
}
```

Para resolver esse problema, na linguagem Java existe uma forma especial de tratamento de exceções para garantir que os recursos abertos sejam sempre fechados. Ela é chamada de *try-with-resources*, que garante que o recurso seja sempre fechado depois de utilizado. Na declaração do `try`, é passado como argumento o recurso que está sendo aberto, e o Java garante que, depois que o bloco é finalizado, o método `close()` da classe do recurso será chamado.

```
public static void main(String[] args) {
    try (BufferedReader reader
```

```
        = new BufferedReader(new FileReader("teste.txt"))) {
        String line;

        while ((line = reader.readLine()) != null) {
            // Processa a linha
            System.out.println(line);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Essa solução é mais segura, pois garante que o recurso será fechado ao final da execução do bloco do `try`, além de ter uma implementação bem mais simples do que a da solução mostrada anteriormente, que precisava ter o bloco `finally` e, dentro dele, também havia um tratamento de exceção.

Mensagens e códigos de erro

Quando problemas acontecerem, é essencial retornar mensagens claras e objetivas para o usuário da aplicação. O usuário precisa entender o problema e saber que ação tomar a partir dessa informação para garantir a confiabilidade da aplicação. Além disso, se a aplicação que está sendo desenvolvida for um serviço REST, é importante atentar-se aos códigos que a aplicação retorna para facilitar o processamento da resposta pelos serviços.

No Spring, existe uma forma bastante interessante de retornar os códigos de erro corretos, que são os `ControllerAdvice`. Nesse tipo de classe, podemos definir um objeto que será retornado caso ocorra uma exceção no código, e também qual o código de erro retornado. No Spring, e na maioria dos frameworks web, por padrão, uma rota REST retorna o código 200, se tudo ocorrer bem, e 500, se ocorrer uma exceção no servidor.

Porém, imagine que temos um código que lança uma exceção chamada `PessoaNaoEncontradaException` em uma rota, como no código

a seguir:

```
@RestController
@RequestMapping("/pessoas")
public class PessoaController {

    @Autowired
    private PessoaRepository pessoaRepository;

    @GetMapping("/{id}")
    public Pessoa getPessoa(@PathVariable Long id) {
        Pessoa pessoa = pessoaRepository.findById(id)
            .orElseThrow(()
                -> new PessoaNaoEncontradaException(id));
        return pessoa;
    }

}
```

Essa rota, caso uma pessoa seja encontrada, retornará os dados da pessoa e um código HTTP 200, mas se nenhuma pessoa for encontrada, a rota retornará um erro 500 e a mensagem "*Internal Server Error*". O correto, porém, seria retornar um código 404, que indica que o recurso não foi encontrado no servidor, e uma mensagem mais amigável, como "*Pessoa não encontrada*".

No Spring, podemos usar uma classe com a anotação `@RestControllerAdvice` para resolver esse problema, como no código mostrado a seguir:

```
@RestControllerAdvice
public class RestExceptionHandler {

    @ExceptionHandler(PessoaNotFoundException.class)
    public ResponseEntity<String> handlePessoaNotFoundException(
            PessoaNotFoundException ex) {
        String mensagem = "Pessoa não encontrada";
        return ResponseEntity
            .status(HttpStatus.NOT_FOUND)
```

```
            .body(mensagem);
    }

}
```

Esse código faz com que, sempre que uma rota lance a exceção `PessoaNotFoundException`, ela seja tratada pelo método `handlePessoaNotFoundException`, que retorna o código 404, definido com o `HttpStatus.NOT_FOUND` e com uma mensagem mais amigável ao usuário.

7.2 Timeouts

Em aplicações que realizam chamadas a recursos externos, como acesso a bancos de dados ou a outras aplicações, podem ocorrer situações em que o serviço chamado tenha problemas para conseguir responder a essa chamada. O cliente, que é o serviço que fez a chamada, pode definir um tempo máximo para que a resposta de uma chamada seja recebida. Caso a resposta não seja recebida dentro desse prazo, o cliente retorna um erro conhecido como ***timeout***.

Embora o *timeout* seja considerado um problema, muitas vezes é preferível receber um *timeout* do que ficar aguardando indefinidamente por uma resposta, ou por um tempo excessivamente longo. Isso ocorre porque, normalmente, as chamadas de rede são bloqueantes, ou seja, a *thread* ou processo que fez a chamada ficará bloqueada enquanto aguarda a resposta.

Vamos considerar uma situação em um servidor web que possui 10 *threads* disponíveis, e a cada requisição para esse servidor é feita uma chamada de rede. Se essa chamada começar a ficar muito lenta, demorando vários minutos para responder, em pouco tempo a aplicação inteira deixará de funcionar, uma vez que todas as *threads* estarão bloqueadas aguardando uma resposta.

Existem vários tipos de *timeouts*. Alguns exemplos são:

- **Conexão:** tempo máximo permitido para estabelecer uma conexão entre dois dispositivos. Se a conexão não for estabelecida dentro desse período, ocorre um *timeout* de conexão;
- **Transmissão:** é o tempo máximo permitido para enviar dados entre dois dispositivos após uma conexão ter sido estabelecida. Se a transmissão não for concluída dentro desse período, ocorre um *timeout* de transmissão;
- **Recebimento:** é o tempo máximo permitido para receber dados de um dispositivo remoto após ter sido estabelecida uma conexão. Se os dados não forem recebidos dentro desse período, ocorre um *timeout* de recebimento;
- **Requisição de conexão:** é o tempo máximo permitido para obter uma conexão de um *pool* de conexões ou estabelecer uma nova conexão com um servidor. Se o *pool* de conexões não responde dentro desse período, ocorre um *timeout* de requisição de conexão.

Esses *timeouts* podem ser utilizados no acesso a qualquer tipo de recurso. Por exemplo, em uma chamada a um banco de dados em uma aplicação com o Spring Boot, podemos configurar o *timeout* de conexão para 5 segundos usando a seguinte propriedade em uma aplicação:

```
spring.datasource.hikari.connection-timeout=5000
```

Ainda no Spring, uma classe que é bastante utilizada para fazer conexões de rede é a `RestTemplate`. Nela, por padrão, os tempos de *timeout* são infinitos, isto é, nunca ocorrerão *timeouts*. Podemos mudar isso configurando os *timeouts* manualmente:

```
HttpComponentsClientHttpRequestFactory customRequestFactory
    = new HttpComponentsClientHttpRequestFactory();
customRequestFactory.setConnectTimeout(15000);
```

```
customRequestFactory.setReadTimeout(30000);
customRequestFactory.setConnectionRequestTimeout(5000);

RestTemplate restTemplate
    = new RestTemplate(customRequestFactory);
```

Para citar um caso real, uma aplicação cliente de um servidor web de vez em quando parava de responder. Depois de muita pesquisa, vimos que o problema era o *timeout* de requisição de conexão. O servidor do cliente fazia várias conexões de rede com diferentes aplicações e, de vez em quando, a aplicação em que eu trabalhava não conseguia iniciar a conexão. Como o *timeout* era infinito, a aplicação parava de funcionar.

7.3 Limitação de taxa (Rate limiting)

Uma implementação extremamente importante em aplicações que disponibilizam serviços para os clientes é o controle da taxa de requisições que eles podem fazer, mecanismo conhecido como ***rate limiting***. Esse mecanismo é fundamental para evitar abusos de requisições por parte de clientes legítimos e também para prevenir ataques DDoS (*Distributed Denial of Service*), nos quais vários dispositivos são utilizados para sobrecarregar um serviço e torná-lo indisponível.

Normalmente, quando um cliente atinge o número máximo de requisições, é retornado o erro HTTP `429 Too Many Requests` (muitas requisições). Existem várias formas de definir um limite para o número de requisições. Por exemplo, podemos definir que um IP pode fazer no máximo 10 requisições por segundo para nossa aplicação, para garantir que não haja nenhum ataque DDoS partindo de um cliente específico.

Outra forma de definir os limites é por contrato. Por exemplo, podemos definir que, no plano que um usuário contratou em nossa

aplicação, ele tem direito a fazer 100 mil requisições por mês para a API. Esse tipo de limite é bastante usado em serviços que são disponibilizados por uma API. Por exemplo, o IMDb, banco de dados sobre filmes, possui uma API que pode ser usada para consultar os dados sobre filmes, atores e diretores. O plano inicial, por exemplo, começa liberando 5 mil requisições por dia para cada usuário, e o número de requisições permitidas vai aumentando nos planos mais caros.

Existem várias formas de implementar o *rate limiting* em uma aplicação. Por exemplo, usando o framework Spring, podemos implementar um componente que usa o Redis para verificar quantas requisições uma `apiKey` faz por segundo. No código a seguir, o método `isAllowed` incrementa em `1` a chave do Redis a cada chamada desse método para a `apiKey` passada como parâmetro. Se for a primeira chamada dessa `apiKey`, é definido que essa chave deve durar 60 segundos no Redis; depois disso, ela expira e a contagem de chamadas para a `apiKey` reinicia. Por fim, é feita a validação se a requisição será autorizada ou não com a verificação `requests <= REQUEST_LIMIT`.

```
@Component
public class RateLimiter {
    private static final int REQUEST_LIMIT = 100;
    private static final long TIME_LIMIT_IN_SECONDS = 60;

    @Autowired
    private RedisTemplate<String, Long> redisTemplate;

    public boolean isAllowed(String apiKey) {
        Long requests = redisTemplate
            .opsForValue()
            .increment(key, 1);

        if (requests == 1) {
            redisTemplate.expire(apiKey,
                TIME_LIMIT_IN_SECONDS, TimeUnit.SECONDS);
        }
```

```
        return requests <= REQUEST_LIMIT;
    }
}
```

Agora podemos usar esse componente em uma rota REST, colocando a chamada para o método `isAllowed` logo no início do método que implementa a rota. Se o método `isAllowed` retorna `true`, o acesso à rota é liberado; caso contrário, pode ser retornado para o usuário um erro avisando que ele já atingiu o limite de requisições e deve tentar novamente mais tarde.

```
@Autowired
private RateLimiter rateLimiter;

@GetMapping("/resource")
public String getResource(
    @RequestHeader(name = "apiKey") String apiKey) {

    if (rateLimiter.isAllowed(apiKey)) {
        // A API key tem permissão para fazer a requisição.
        // Execute sua lógica de negócios aqui.
        return "Recurso retornado com sucesso!";
    } else {
        // A API key atingiu o limite de taxa de requisições.
        // Retorne uma resposta adequada ao cliente.
        return "Limite de taxa de requisições atingido.
            Tente novamente mais tarde.";
    }
}
```

Aqui, por simplificação, o componente `RateLimiter` foi usado diretamente na rota. Isso pode não ser uma boa ideia, pois teríamos que replicar esse código em todos os métodos que implementam rotas REST em nossa aplicação. Para evitar isso, teríamos algumas opções:

- ***Api-gateway*:** fazer a validação do *rate limiting* em um *Api-gateway*, que é uma aplicação que fica antes das rotas e serve

para fazer processamentos que podem ser feitos antes da execução da rota — por exemplo, o próprio *rate limiting* e a validação da `apiKey` ;
- **Filtro:** todo framework web, como o Spring, possui uma implementação de filtros HTTP. Ele é executado antes e depois do processamento dos recursos da aplicação, permitindo a realização das mesmas tarefas de um *Api-gateway*, como autenticação, autorização, registro de logs e *rate limiting*;

- **Anotação:** podemos criar uma anotação que faz o *rate limiting*. Por exemplo, a anotação poderia ser `@RateLimit` , e todo método que receber essa anotação faz a verificação se o usuário pode ou não acessar a rota.

7.4 Monitoramento

Implementar o monitoramento nas aplicações nem sempre é simples, mas, para facilitar as coisas, existem diversas bibliotecas prontas para coletar dados da aplicação e disponibilizá-los para ferramentas de visualização e análise de dados. Alguns exemplos são o DataDog (https://www.datadoghq.com/) e o Dynatrace (https://www.dynatrace.com/), que são soluções pagas e fáceis de plugar em qualquer tipo de aplicação.

Felizmente, existem também soluções livres que podem ser usadas sem nenhum custo. Por exemplo, em uma aplicação Java que utiliza o Spring Boot, é possível utilizar a biblioteca `spring-boot-starter-actuator` , que automaticamente cria vários indicadores sobre a saúde da aplicação. Para utilizar essa biblioteca, basta adicionar a dependência em um gerenciador de dependências como o Maven.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Nessa biblioteca, por padrão, apenas algumas métricas são disponibilizadas, mas podemos mudar isso adicionando a configuração `management.endpoints.web.exposure.include=*`, que adicionará todas as métricas possíveis, ou escolher algumas das métricas, por exemplo, adicionando o trecho `management.endpoints.web.exposure.include=env,health,metrics`. Isso é feito porque a geração de métricas pode ser pesada, o que influencia no desempenho da aplicação. Portanto, devemos habilitar apenas as métricas que realmente são necessárias.

Agora, automaticamente a aplicação disponibilizará um *endpoint* no endereço `http://localhost:8080/actuator`, que pode ser acessado em um browser e que mostrará todas as métricas que podem ser encontradas. Essa URL retornará um JSON onde algumas das métricas podem ser encontradas:


```
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    },
    "env": {
      "href": "http://localhost:8080/actuator/env",
      "templated": false
    },
    "metrics-requiredMetricName": {
      "href":
"http://localhost:8080/actuator/metrics/{requiredMetricName}",
      "templated": true
    },
    "metrics": {
      "href": "http://localhost:8080/actuator/metrics",
      "templated": false
```

```
    }
  }
}
```

Podemos acessar agora a URL `http://localhost:8080/actuator/metrics`, que mostra uma lista com todas as métricas que estão disponíveis, como tempo de inicialização da aplicação, consumo de memória e número de *threads*.

Por fim, podemos acessar o valor de uma métrica disponível, como no exemplo o tempo de inicialização da aplicação na URL `http://localhost:8080/actuator/metrics/application.started.time`. A resposta da chamada desse *endpoint* retornará um JSON com diversas informações sobre a inicialização da aplicação, como no exemplo a seguir:

```
{
  "name": "application.started.time",
  "description": "Time taken (ms) to start the application",
  "baseUnit": "seconds",
  "measurements": [
    {
      "statistic": "VALUE",
      "value": 4.521
    }
  ],
  "availableTags": [
    {
      "tag": "main.application.class",
      "values": [
        "com.santana.arquitetura.ArquiteturaApplication"
      ]
    }
  ]
}
```

Obviamente, ficar acessando esses *endpoints* manualmente é inviável. Por isso, existem outras ferramentas complementares que servem para coletar as informações em tempos reais e facilitar a visualização dos dados. Duas ferramentas muito usadas para isso são o Prometheus e o Grafana. O **Prometheus** é uma ferramenta de coleta de métricas, na qual é possível configurar um processo que periodicamente coletará e armazenará as métricas em um *endpoint*, como mostrado anteriormente. Já o **Grafana** é uma ferramenta para a visualização de dados, e ela se conecta facilmente ao Prometheus.

Conclusões

Neste capítulo, vimos que a confiabilidade de uma aplicação depende de diversos fatores. Inicialmente, precisamos deixar o nosso código o mais robusto possível para evitar que falhas gerem um efeito catastrófico na aplicação. Para isso, podemos usar técnicas como o tratamento de exceções. Além disso, no acesso a recursos, precisamos nos certificar de que todos os recursos sejam liberados ao fim da utilização.

Ainda no código, precisamos instrumentalizar a nossa aplicação para que ela gere métricas sobre a sua saúde, permitindo monitorar possíveis problemas. Porém, não adianta apenas deixar a aplicação robusta e confiável: a infraestrutura também precisa atender a esses dois requisitos. Por isso, no próximo capítulo, veremos diversas técnicas que podemos aplicar na infraestrutura para melhorar a confiabilidade de nossas aplicações.

Capítulo 8
Confiabilidade na infraestrutura

Vimos que a confiabilidade de uma aplicação depende bastante do código, mas não adianta ter um código robusto, que trata corretamente as falhas, e ter problemas na infraestrutura da aplicação. Uma aplicação que necessita de alta disponibilidade deve possuir uma infraestrutura bem projetada, de modo que quaisquer problemas eventuais no hardware que executa a aplicação não a interrompam ou a tornem inacessível aos clientes.

Existem diversas técnicas para diminuir a chance de uma aplicação ficar indisponível. A principal é o balanceamento de carga, no qual a mesma aplicação é executada em diversas máquinas diferentes (muitas vezes, em localidades geográficas diferentes) para que a aplicação esteja sempre no ar. No entanto, além de garantir a disponibilidade contínua da aplicação, é indispensável que existam mecanismos de recuperação para lidar com possíveis problemas na aplicação. Diversas técnicas são empregadas nesse contexto, sendo o backup uma das mais comuns e essenciais para qualquer aplicação.

Neste capítulo, vamos estudar algumas técnicas que são essenciais para qualquer aplicação ter uma alta confiabilidade, seja na sua execução ou para que ela consiga se recuperar de falhas sem efeitos colaterais.

8.1 Balanceamento de carga

Já falamos sobre balanceamento de carga antes, quando discutimos o desempenho da aplicação. No entanto, o balanceamento de carga é não apenas útil para o desempenho, mas também essencial para

garantir a alta disponibilidade da aplicação. Isso ocorre porque é certo que uma máquina parará de funcionar em algum momento, seja devido a um defeito de hardware, por exemplo, ou simplesmente por um problema de energia no prédio onde a máquina está localizada, resultando no seu desligamento.

Portanto, confiar em apenas uma máquina é muito arriscado, mesmo que ela esteja em um local extremamente seguro e possua recursos da melhor qualidade possível. Para resolver esse problema, podemos implementar o balanceamento de carga da mesma forma discutida no capítulo 5, *Desempenho e escalabilidade na infraestrutura*. Como mencionado anteriormente, basicamente criamos várias instâncias da mesma aplicação e colocamos um balanceador de carga na frente, responsável por distribuir as requisições entre essas instâncias.

Pensando em *confiabilidade*, em um primeiro passo, o balanceamento de carga funciona exatamente igual ao que fizemos no capítulo 5: podemos criar duas ou mais máquinas idênticas em um servidor de nuvem que executam a aplicação, e o balanceador de carga distribui as requisições entre elas. Isso é o suficiente para a grande maioria das aplicações. No entanto, existem aplicações em que a disponibilidade deve ser de praticamente 100%, sendo um requisito essencial. Se a aplicação precisar de um nível de confiabilidade ainda mais alto, precisaremos de uma estratégia ainda mais complexa, que tenha máquinas distribuídas geograficamente em diferentes regiões de um país ou até mesmo do mundo. A maior parte dos grandes provedores de computação em nuvem possui ferramentas para esse tipo de implantação.

A AWS, por exemplo, é dividida em várias *regiões* distribuídas globalmente, e cada uma dessas regiões é composta por mais de uma **Zona de Disponibilidade** (*Availability Zones* - AZ). Uma das regiões que pode ser usada para a utilização de recursos na AWS é São Paulo (*sa-east-1*), e dentro dessa região existem três AZs: *sa-east-1a*, *sa-east-1b* e *sa-east-1c*. Cada uma dessas AZs é um *data center* diferente, e eles estão em diferentes localidades, ligados em

diferentes redes elétricas e de comunicação. Fazer a distribuição da carga da aplicação em máquinas que estão em diferentes AZs é uma forma importante de aumentar a disponibilidade de uma aplicação em sua infraestrutura. Então, é possível criar uma instância do EC2 em cada uma das três regiões e configurar um *Load Balancer* para distribuir a carga entre essas três instâncias.

Essa distribuição entre as diferentes AZs já garante uma altíssima disponibilidade, o que pode já ser suficiente para a grande maioria das aplicações. Porém, ainda pode acontecer uma indisponibilidade geral em uma região de um provedor de nuvem, o que já aconteceu algumas vezes. Então, é preciso elevar mais ainda o nível da confiabilidade da aplicação e distribuir os recursos entre diferentes regiões do provedor na nuvem. Serão necessários recursos ainda mais avançados, pois o *Load Balancer* da AWS funciona apenas para instâncias que estão na mesma região (podem estar em diferentes AZs dentro da mesma região).

Para isso, a AWS possui um serviço chamado **Global Accelerator** (GA), que permite o balanceamento de carga e o roteamento entre instâncias hospedadas em diferentes regiões da AWS. Além disso, o GA também faz o direcionamento de tráfego dos usuários para os servidores e serviços mais próximos dele, melhorando o desempenho e a latência de uma aplicação que é usada por usuários de todo o mundo. Obviamente, um serviço desse tipo vai ter um custo alto, pois, além do próprio GA, será necessário manter uma redundância dos serviços em diversas regiões do globo.

8.2 Recuperabilidade

Como já falado nos dois capítulos anteriores, um dia sua aplicação com certeza falhará. Porém, temos que assegurar que, caso isso aconteça, conseguiremos reestabelecer os serviços sem grandes consequências para a operação da aplicação. Por isso, temos que

garantir que a configuração da aplicação seja a mais otimizada possível, para que o processo de recuperação seja rápido e fácil.

Veremos algumas formas de garantir a recuperabilidade de uma aplicação, tais como realizar backups periódicos, configurar a reinicialização automática dos serviços e definir um plano de contingência para definir o que fazer em caso de problemas em uma aplicação.

Backups

O primeiro e talvez mais conhecido mecanismo para a recuperabilidade de sistemas é manter backups de tudo que pode precisar ser recuperado. Sem dúvida, o principal backup que deve ser mantido é o do banco de dados da aplicação, pois normalmente os dados não poderão ser gerados novamente. É importante que haja uma política de backup de dados, que define, por exemplo, qual a periodicidade dos backups (de hora em hora, uma vez por dia etc.), por quanto tempo os backups serão mantidos e qual o algoritmo de geração deles.

A **periodicidade dos backups** depende do tipo de sistema em execução. Por exemplo, em um sistema bancário com milhares de transações contínuas, onde cada transação é crucial para manter a consistência da aplicação, é necessário fazer backups praticamente o tempo todo. Além dos backups, essas aplicações geralmente registram logs das transações, o que permite a reconstrução das transações perdidas em caso de falhas.

Por outro lado, em aplicações mais simples, com menor carga, como um sistema de notas em uma universidade no qual as notas são digitadas apenas em determinados períodos durante um semestre, backups a cada hora ou até mesmo diários podem ser suficientes. Então, o intervalo de backup depende bastante da frequência de atualizações e da importância de não perder os dados em caso de falha.

O período de manutenção dos backups é bastante importante por dois motivos: primeiro, porque o tamanho dos arquivos que precisam ser mantidos pode ser bem grande, o que pode gerar custos absurdos para a manutenção deles, tanto em infraestrutura própria quanto em provedores de nuvem. Além disso, praticamente todos os lugares hoje possuem leis de proteção de dados, como a LGPD no Brasil e a GDPR da União Europeia, que garantem vários direitos aos usuários das aplicações sobre seus dados, inclusive os que estão em backups.

Ainda com relação ao tamanho dos arquivos, existem diversos algoritmos para a geração de backups:

- **Backup completo:** cria uma cópia completa de todos os dados todas as vezes que o backup é gerado. É simples e fácil de restaurar, mas consome muito espaço de armazenamento. Neste caso, normalmente se guarda apenas os últimos backups gerados, pois apenas um arquivo é capaz de restaurar todos os dados;

- **Backup incremental:** nesse método, são copiados apenas os dados do último backup gerado, então sempre há um backup completo inicial e, depois, cada novo backup tem apenas as operações geradas desde o último backup. Esse método usa muito menos espaço de armazenamento, mas a restauração é mais demorada e complexa, pois será necessário ler o backup inicial completo e, depois, todos os incrementais;

- **Backup diferencial:** esse método tem uma ideia semelhante ao incremental, mas, neste caso, é gerado um backup completo inicial e, depois, cada novo backup salva as diferenças que existem deste backup inicial. A restauração é mais rápida do que no backup incremental, pois apenas o backup completo e o último backup diferencial são necessários para restaurar os dados.

Reinicialização automática

É importante garantir que as aplicações sejam reinicializadas automaticamente quando uma máquina ou um cluster é reinicializado, e existem inúmeras formas de garantir isso. Por exemplo, no Linux, se for usado o `systemd` para configurar a inicialização das aplicações, podemos garantir que elas sejam inicializadas automaticamente sempre que o sistema operacional for reinicializado. O script a seguir mostra um exemplo de como uma aplicação Java pode ser configurada com o `systemd`:

```
[Unit]
Description=Meu app
After=network.target

[Service]
Type=simple
ExecStart=java -jar /path/app.jar
WorkingDirectory=/path
Restart=always

[Install]
WantedBy=multi-user.target
```

A maioria dos campos desse código é autoexplicativa, mas o mais importante no exemplo é o campo `Restart`, que indica que o serviço deve ser sempre reinicializado. Se a aplicação utiliza contêiner, também é possível configurar a reinicialização automática dos serviços. Por exemplo, no Docker, podemos utilizar a *flag* `--restart` para garantir que um contêiner seja reinicializado automaticamente sempre que ele parar de executar. Essa *flag* possui quatro opções, que são:

- **`no`**: nunca reinicia um contêiner automaticamente;
- **`always`**: sempre reinicia o contêiner. Quando ele é parado manualmente, como com o comando `docker stop`, o contêiner é reiniciado caso o Docker também seja reiniciado;

- **`unless-stopped`** : parecido com o `always` , mas se o contêiner estiver parado quando o Docker for reiniciado, o contêiner não é reiniciado;

- **`on-failure[:max-retries]`** : reinicia o contêiner se ele terminar com falha. Podemos definir o número máximo de tentativas de reinicialização. Por exemplo, para duas tentativas, a configuração é `on-failure:2` .

O comando a seguir mostra como pode ser executado um contêiner do Docker rodando o Redis e que será reinicializado usando a opção `always` :

```
docker run --restart always redis
```

Planos de contingência

Um problema que ainda atinge muitas empresas, especialmente empresas pequenas e startups, é ter uma pessoa que detém quase todo o conhecimento sobre uma determinada aplicação. Normalmente isso acontece quando apenas uma pessoa inicia um projeto, e somente ela detém o conhecimento de como tudo funciona. Esse cenário pode levar ao caso de que, quando uma falha acontecer na aplicação, apenas essa pessoa conseguirá resolver os problemas.

Para solucionar isso, toda aplicação deve ter uma série de documentos que define uma sequência de passos a serem seguidos quando uma falha acontece.

O **plano de contingência** é um desses documentos, e ele deve descrever as ações e medidas a serem tomadas em caso de falhas ou interrupções que possam afetar a operação normal de uma aplicação. Ele é um componente importante do planejamento de continuidade de negócios e tem como objetivo minimizar o impacto de tais eventos e facilitar a recuperação rápida e eficiente do sistema. Normalmente esse documento é obrigatório em auditorias

de qualidade de software como ISO e SOC (falaremos mais dessas auditorias no capítulo 9, *Segurança*).

Alguns elementos que normalmente são incluídos no plano de contingência são:

- **Cenários de falhas**: identificar e descrever os possíveis cenários de falhas, especialmente os que têm maior probabilidade de ocorrência de problemas dependendo do tipo da aplicação e das características de sua infraestrutura. Alguns exemplos desses cenários são falhas de hardware, falhas de software, desastres naturais e ataques cibernéticos. Cada cenário deve ser analisado em termos de probabilidade de ocorrência e impacto no sistema;

- **Responsabilidades e comunicação**: definir quais são as equipes envolvidas em cada tipo de falha que pode acontecer na aplicação e quais os canais de comunicação disponíveis. Por exemplo, para uma falha de infraestrutura, deve ser acionada a equipe de DevOps, que tem um telefone como canal de comunicação; em uma falha de software, a equipe de desenvolvimento também deve ser acionada. Além disso, também é importante identificar os indivíduos responsáveis por cada equipe;

- **Procedimentos de recuperação**: descrever os diversos possíveis procedimentos para a recuperação de cada cenário de falha. Esses procedimentos devem ser suficientemente detalhados para que qualquer membro da equipe responsável pela falha consiga fazer a recuperação da aplicação. O detalhamento pode incluir instruções passo a passo, configurações de backup, configurações de rede, procedimentos de restauração de dados e scripts de recuperação;

- **Infraestrutura de backup**: especificar os sistemas de backup necessários para garantir a disponibilidade e integridade dos dados. Isso envolve a definição de políticas de backup, como

frequência, local de armazenamento e mecanismos de recuperação.

Ter um plano de contingência adequado em engenharia de software é fundamental para garantir a resiliência e a continuidade das operações do sistema, minimizando o tempo de inatividade e o impacto em caso de falhas. Essa documentação deve ser desenvolvida em colaboração com profissionais de infraestrutura, segurança e gestão de projetos, levando em consideração as especificidades do sistema e as necessidades da organização.

8.3 Deployability

Eu evitei usar termos em inglês no livro todo, mas não consegui encontrar uma boa tradução para ***deployability*** (talvez *implantabilidade*), que se refere ao requisito não funcional que destaca que uma aplicação deve ser fácil de ser implantada em um servidor. Esse requisito é extremamente importante, porque o momento da troca de versão de uma aplicação é um dos mais arriscados em relação à confiabilidade de uma aplicação. Uma implantação mal-feita e mal testada pode deixar uma aplicação fora do ar, e verificar o que causou o erro na reinicialização da aplicação nem sempre é simples.

Integração e implantação contínuas

Um conceito bastante importante ligado à implantação de uma aplicação são os fluxos de CI (*Continuous Integration*) e CD (*Continuous Deployment*) feitos por ferramentas como o GitHub ou o Jenkins. No fluxo para a implantação de uma aplicação, podemos definir diversos rituais que, apesar de serem burocráticos, podem diminuir muito o risco de problemas durante a atualização de versões de aplicações nos servidores.

Existem diversas formas de fazer o CI/CD em uma empresa. Uma que eu gosto de utilizar é a criação de ambientes separados e específicos para determinados fins. Por exemplo, uma empresa pode manter quatro ambientes, um de desenvolvimento, um de testes, um de homologação e um de produção. Cada um desses ambientes possui um propósito específico:

- **Ambiente de desenvolvimento**: utilizado para testes do time de desenvolvimento, é um ambiente não muito confiável, pois ele pode receber novas funcionalidades que ainda estão em desenvolvimento;
- **Ambiente de testes**: é onde são executados os testes de QA (*Quality Assurance*). Ainda não é totalmente confiável, pois os testes da equipe de qualidade ainda estão em andamento, mas esse ambiente normalmente possui funcionalidades que o time de desenvolvimento considera prontas;
- **Ambiente de homologação**: depois que os testes de QA são executados, normalmente a funcionalidade vai para um ambiente de homologação, que normalmente é um ambiente quase tão confiável ao de produção. Nesse ambiente, o time de produtos valida se as funcionalidades foram implementadas corretamente;
- **Ambiente de produção**: por fim, o ambiente de produção é onde a aplicação é executada de verdade, tendo acessos de clientes. Esse ambiente tem que ser o mais confiável possível.

Obviamente podem existir variações: por exemplo, uma empresa pode não ter o ambiente de homologação e fazer as validações diretamente no ambiente de testes. Cada um desses ambientes pode ser controlado por uma *branch* separada no Git, assim conseguimos controlar exatamente qual versão do software está sendo executada em cada um dos ambientes. Além disso, é possível usar ferramentas de CD para fazer a implantação do sistema.

Algumas dessas ferramentas são o GitHub Actions e o já citado Jenkins.

Vamos analisar como podemos construir um fluxo para a implantação de uma aplicação usando o GitHub Actions. Para isso, precisamos criar um arquivo YAML na pasta `.github/workflows` e ele deve ser configurado em três seções principais: a primeira contém o nome do workflow; a segunda define quais ações vão disparar o workflow, e em quais *branches*; e a terceira define as ações que serão executadas. No primeiro trecho são mostradas essas três partes, que definem o nome, a execução do workflow sempre que houver um `push` na *branch* `main`, e o início da seção das ações, que indica que ele será executado em uma máquina Ubuntu:

```
name: Exemplo de Build

on:
  push:
    branches:
      - main

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
```

Depois, podemos definir vários passos (*steps*), e cada um vai executar uma ação dentro da máquina que executará o *workflow*. No nosso exemplo, as três primeiras ações fazem algumas configurações na máquina, que são:

- Fazer o *checkout* do projeto;
- Instalar o Java na versão 17;
- Instalar o Docker.

```
steps:
  - name: Checkout do código
    uses: actions/checkout@v2

  - name: Instala e configura o Java
```

```
    uses: actions/setup-java@v2
    with:
      java-version: 17

  - name: Set up Docker
    uses: docker/setup-buildx-action@v1
```

Com isso, podemos começar a executar alguns comandos nessa máquina, cujas ações são:

- Construir o projeto Java com o Maven;
- Construir a imagem do Docker;
- Fazer o login na Amazon ECR (que é um repositório de imagens do Docker na AWS);
- Enviar a imagem do Docker construída para o ECR;
- Fazer login no servidor que executa a aplicação via SSH e inicia a aplicação. Veja que o login está sendo feito com a chave SSH direto no código; isso é uma falha de segurança, mas veremos como resolver isso no capítulo 11, *Segurança na infraestrutura*.

```
- name: Faz o build do projeto
  run: mvn clean package

- name: Constrói a imagem do Docker
  run: docker build -t app .

- name: Log in to Amazon ECR
  uses: aws-actions/amazon-ecr-login@v1

- name: Push Docker image to Amazon ECR
  run: docker push your-docker-image-name

- name: Deploy to EC2 instance
  run: |
    ssh -o StrictHostKeyChecking=no -i chave-ssh ec-2user@ip-address
    docker run -d app
```

Rollbacks

Também relacionada à implantação de uma aplicação está a facilidade de fazer ***rollbacks***, isto é, voltar para a versão anterior caso algum problema aconteça durante a implantação ou durante os testes de aceitação feitos no ambiente de produção. Isso tem que ser um processo simples e rápido, senão uma aplicação pode ficar muito tempo fora do ar caso haja algum problema durante uma implantação (o que é bastante comum).

Quando os arquivos são implantados manualmente em um servidor, a forma mais simples de garantir o *rollback* é salvar uma cópia do arquivo antigo e depois fazer a substituição. Além disso, em uma implantação, muitas vezes precisamos executar scripts em bancos de dados, por isso, além dos scripts que fazem a operação, é necessário também fazer os scripts que revertem as operações caso necessário.

Fazer isso manualmente é trabalhoso e também arriscado: a chance de uma falha humana acontecer em um processo desse é sempre bem maior do que se o processo estiver totalmente automatizado. Por isso, atualmente, com as tecnologias usadas para o controle de versão, automações e com o uso de contêineres, configurar uma estratégia de *rollback* é uma tarefa relativamente fácil.

Existem várias opções para implantar isso em uma empresa, e uma das ferramentas mais usadas é o próprio Git. Por exemplo, se for utilizado um fluxo de CI/CD que usa uma *branch* do Git para fazer a implantação em um servidor, basta remover o código que foi adicionado desta *branch* que automaticamente a versão será revertida. Para facilitar mais as coisas, a interface gráfica do GitHub permite a reversão de um `pull request` inteiro, como mostra a figura a seguir:

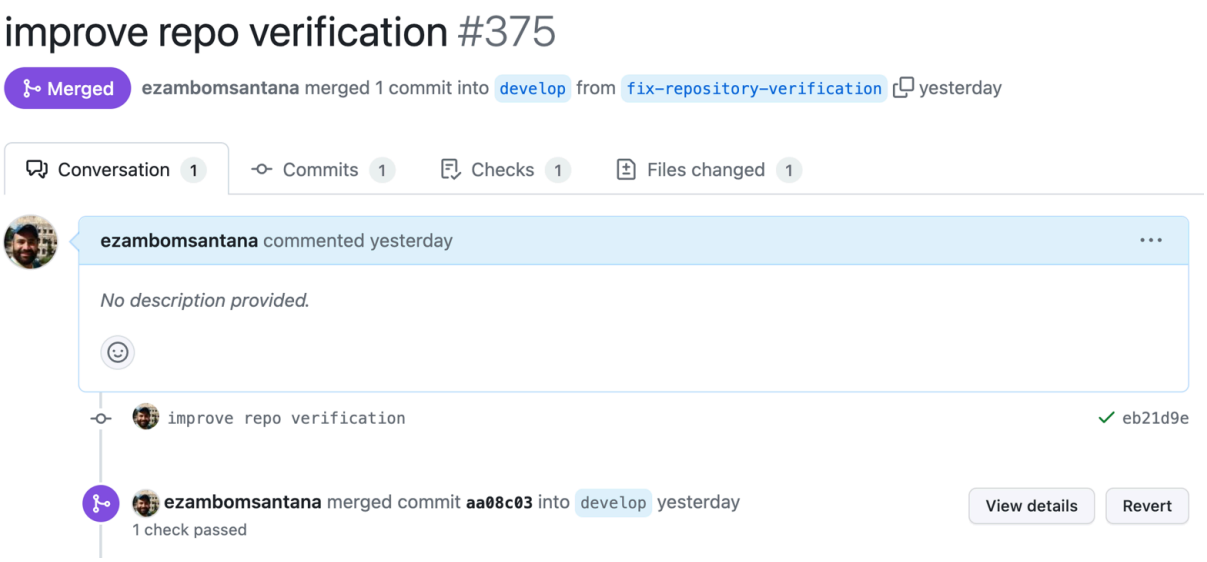


Figura 8.1: Reversão de Código no GitHub

No canto inferior direito da imagem, existe um botão chamado `revert`. Esse processo abre um outro `pull request`, fazendo a reversão de tudo que foi feito no `pull request` original. Com isso, fazer a reversão diretamente no código fica bastante simples.

Infraestrutura como código

Outra técnica bastante utilizada na implantação e que ajuda bastante na confiabilidade é fazer a definição da infraestrutura da aplicação em código. Com ela, é possível evitar erros humanos na criação e na manutenção da infraestrutura da aplicação. Ela também facilita a reprodutibilidade de vários ambientes, garantindo que todos funcionem mais ou menos da mesma forma e possuam os mesmos recursos.

É possível fazer a definição da infraestrutura usando uma linguagem de programação. Na AWS, por exemplo, é possível realizar diversas ações na infraestrutura usando scripts que podem ser desenvolvidos em linguagens como Python, entre outras. O script a seguir mostra como pode ser feita a criação de uma máquina no EC2 com Python. No script, também é possível esperar a confirmação da criação da máquina e depois recuperar algumas informações dela. Veja:

```python
import boto3

# Configure as credenciais da AWS (substitua pelos seus próprios valores)
aws_access_key_id = 'SEU_ACCESS_KEY'
aws_secret_access_key = 'SEU_SECRET_KEY'
region_name = 'us-east-1'

# Crie uma sessão do boto3 usando as credenciais configuradas
session = boto3.Session(
    aws_access_key_id=aws_access_key_id,
    aws_secret_access_key=aws_secret_access_key,
    region_name=region_name
)

# Crie o recurso EC2 usando a sessão
ec2 = session.resource('ec2')

# Crie a instância EC2
instance = ec2.create_instances(
    ImageId='AMI_ID',         # Imagem usada para a criação da máquina
    InstanceType='t2.micro',  # Tipo da instância
    MinCount=1,
    MaxCount=1
)[0]  # Pegamos a primeira (e única) instância

# Espere até que a instância esteja em estado 'running'
instance.wait_until_running()

# Recupere informações sobre a instância criada
instance.load()

print(f"Instância EC2 criada com ID: {instance.id}")
print(f"Endereço IP público: {instance.public_ip_address}")
```

Além disso, existem ferramentas específicas, como o **Terraform**, com o qual é possível definir todos os elementos da infraestrutura usando uma linguagem própria da ferramenta. O exemplo a seguir

mostra um script que pode ser usado para criar uma máquina no EC2 usando essa ferramenta. Nele, basicamente temos que definir o `provider`, que é a AWS (funciona no GCP e no Azure também), e uma lista de `resources`, no caso o `aws_instance`, que indica que será criada uma máquina no `EC2`. É possível fazer diversas configurações para a criação dessa máquina, entre elas a imagem que será usada e o tipo da instância:

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example_instance" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Conclusões

Existe uma infinidade de ferramentas e tecnologias que podem ser utilizadas para aumentar a confiabilidade da infraestrutura de uma aplicação. Neste capítulo, passamos por algumas das mais fundamentais, que são:

- O *balanceamento de carga*, para garantir que, em caso de falha, não exista apenas um ponto de execução da aplicação;
- A *recuperabilidade*, que define diversas práticas que devem ser realizadas pelas equipes dentro de uma empresa para garantir que, quando alguma coisa de errado acontecer (e vai acontecer), seja possível recuperar a aplicação com o mínimo possível de efeitos colaterais;
- A *deployability*, que visa facilitar a tarefa de implantação de uma nova versão, um dos momentos mais críticos para qualquer aplicação.

Um requisito que também é bastante relacionado à confiabilidade é a **segurança** da aplicação, pois uma aplicação que tem problemas de segurança provavelmente também não é muito confiável. Porém, segurança por si só é um assunto tão complexo como a confiabilidade, por isso vamos discutir esse outro requisito fundamental para qualquer aplicação nos próximos três capítulos.

Parte 4: Segurança

Capítulo 9
Segurança

Agora que nossas aplicações já estão escaláveis e confiáveis, precisamos também garantir que elas sejam seguras. Dos três requisitos, talvez esse seja o mais complicado e abrangente, pois a segurança de uma aplicação possui vários significados. O mais óbvio talvez seja o controle de usuários, que significa permitir que apenas usuários cadastrados e autorizados possam acessar certas partes de um sistema. Esses controles são chamados de *autenticação* e *autorização*.

Porém, isso é apenas uma pequena parte do que devemos fazer para garantir a segurança de nossa aplicação. Por exemplo, temos que assegurar que todos os nossos servidores estejam protegidos e que apenas usuários autorizados possam acessar essas máquinas. Dados importantes da aplicação, como senhas de banco de dados de produção, não devem estar presentes no código que está no repositório. Dados privados de usuários, como endereços, não devem ser escritos nos logs da aplicação, aos quais várias pessoas funcionárias da empresa podem ter acesso.

Também é essencial monitorar ativamente a aplicação para garantir que nenhuma medida de segurança seja violada e para detectar qualquer comportamento anormal que possa estar relacionado a um ataque à aplicação. Por isso, é necessário implementar boas práticas de segurança em todas as camadas da aplicação, desde a infraestrutura até o código. Ao adotar uma abordagem abrangente de segurança, as organizações podem proteger seus sistemas e dados e garantir a confiança dos usuários.

Nos próximos três capítulos, discutiremos algumas das técnicas mais importantes para garantir a segurança da aplicação e como podemos

implementar essas técnicas tanto diretamente no código da aplicação quanto em sua infraestrutura.

9.1 Controle de acesso

Um dos requisitos mais importantes de segurança é o **controle de acesso** aos recursos de uma aplicação. Existem diversos níveis de controle de acesso. Por exemplo, temos o controle de acesso básico em uma aplicação, no qual é solicitado ao usuário seu e-mail e sua senha para que possa acessar a aplicação. Com isso, dentro de um sistema, apenas os dados a que esse usuário tem acesso serão exibidos.

No entanto, também temos o controle de acesso na infraestrutura da aplicação. O banco de dados da aplicação é uma das áreas mais sensíveis relacionadas à segurança, pois nele existem diversos dados dos usuários. Em um e-commerce, por exemplo, existem informações sensíveis como o endereço, meios de pagamentos e até o histórico de compras de um usuário. Quem pode ter acesso a esses dados? Com certeza, não pode ser um dado aberto para que qualquer funcionário de uma empresa possa acessar.

Existem diversos outros recursos que devem ter um controle de acesso forte, como o repositório com o código-fonte de uma aplicação. Alguns devem ter acesso apenas de leitura no código, outros devem ter acesso de escrita e outros ainda com acesso de administrador, podendo alterar as configurações do repositório. Por isso, o controle de acesso é um dos requisitos mais críticos em qualquer tipo de aplicação.

Autenticação

A **autenticação** é o processo de garantir a identidade do usuário, sendo o mecanismo mais utilizado para isso o de pedir o nome do usuário (ou um e-mail) e sua senha. Porém, dependendo do nível de segurança que uma aplicação necessita, apenas pedir o usuário e senha pode ser insuficiente, pois esse método possui várias brechas — por exemplo, usuários que usam senhas extremamente simples ou ataques que interceptam a comunicação entre o cliente e o servidor.

Por isso, existem diversos outros mecanismos de autenticação. Um exemplo bastante utilizado na infraestrutura das aplicações são as chaves de acesso SSH (Secure Shell). O SSH é um protocolo seguro para a comunicação entre computadores, permitindo controlar uma máquina remotamente. Apesar de ser possível usar SSH com um usuário e senha, o mais comum é usar uma chave SSH, que é um algoritmo de criptografia assimétrica.

A criptografia de chave assimétrica envolve o uso de um par de chaves: uma chave privada e uma chave pública. A chave privada é mantida em segredo pelo proprietário e a chave pública é compartilhada com outros, especificamente os servidores remotos aos quais se deseja conectar. Esse mecanismo é muito mais seguro que uma autenticação simples usuário/senha e é muito utilizado para o acesso a servidores e aos repositórios de código-fonte.

Autorização

Além de identificar um usuário com a autenticação, é necessário verificar também se ele tem acesso a um recurso. Assim como a autenticação, a **autorização** também é importante em várias partes diferentes da aplicação. Por exemplo, em um e-commerce, é necessário ter os vários papéis que podem acessar a aplicação. Nela, um cliente tem permissões apenas para buscar produtos e realizar compras; já um operador pode editar e cadastrar produtos na plataforma, e um(a) atendente da área do consumidor pode acessar os dados do pedido para verificar a reclamação de um cliente.

Isso normalmente é implementado via papéis na aplicação. Quando um usuário faz o login em uma aplicação, é verificado o seu papel (*role*), e normalmente as aplicações têm visões totalmente diferentes para os diferentes papéis. Praticamente todas as aplicações comerciais precisam implementar algum nível de autorização.

No entanto, além da autorização implementada diretamente na aplicação, também é necessário definir esse tipo de autorização na infraestrutura. Na autenticação, definimos quais usuários podem acessar a infraestrutura da aplicação, como servidores e banco de dados, mas agora precisamos definir também quais são os níveis de acesso que um usuário vai ter na infraestrutura. Por exemplo, um desenvolvedor pode ter acesso apenas de leitura em um servidor de produção, enquanto um líder técnico pode ter direito de escrita para poder reiniciar uma aplicação.

Além dos servidores, isso acontece em várias outras partes da infraestrutura de uma aplicação. Alguns exemplos são:

- **Banco de dados:** quais usuários podem acessar as tabelas de um banco de dados? Desenvolvedores podem acessar tabelas que não contenham dados sensíveis de usuários?
- **Repositório de código:** todos os usuários do repositório podem acessar todos os repositórios? Ou devem ter acesso apenas aos repositórios em que estejam trabalhando? O acesso deve ser apenas de leitura ou pode ser de escrita também?
- **CI/CD:** quais usuários podem inicializar uma execução de um fluxo de CI/CD? Deve haver alguma autorização para que esses fluxos sejam inicializados?

9.2 Confidencialidade

Por mais simples que uma aplicação seja, com certeza ela vai armazenar e trafegar algumas informações sensíveis. O exemplo mais simples são os dados básicos de um usuário, como o e-mail e a senha, mas dependendo do tipo do sistema, podem existir dados muito mais sensíveis, como endereços, meios de pagamento e dados de saúde. Além disso, toda aplicação tem seus próprios dados sensíveis, como o usuário e senha de acesso ao banco de dados e chaves de acesso a outras aplicações.

Por isso, é necessário garantir a confidencialidade dos dados na aplicação. A primeira forma de garantir isso já foi discutida, que é controlar o acesso aos dados. Porém, nem sempre será possível proibir o acesso aos dados, e sempre haverá tentativas de acessos indevidos a esses dados. Por isso, a forma mais comum para diminuir a chance de dados indevidos é usar algoritmos criptográficos para esconder os dados reais. Dessa forma, mesmo que alguém não autorizado consiga acessar os dados, não vai conseguir lê-los.

A confidencialidade dos dados deve ser feita tanto na aplicação em si — com a implementação de algoritmos criptográficos para armazenar ou transmitir dados importantes, como senhas, números de cartões de crédito e chaves de acesso — quanto na infraestrutura — como a utilização do protocolo HTTPS em vez do HTTP. Nesse protocolo, os dados entre o cliente e o servidor são criptografados, evitando que ataques que interceptam a troca de dados entre cliente e servidor consigam ler os dados trocados.

Ainda na infraestrutura, é importante ter mecanismos para que as senhas que a aplicação usa, como senhas de banco de dados e chaves para acessos a sistemas externos, sejam aplicadas sem que esses dados tenham que estar no código ou que uma pessoa desenvolvedora específica tenha que conhecer essas senhas. Existem vários mecanismos para isso: as aplicações de CI/CD, por exemplo, conseguem armazenar segredos que são aplicados apenas quando a aplicação está sendo implantada no servidor.

9.3 Monitoramento

Já comentamos sobre o *monitoramento* da aplicação para melhorar a confiabilidade de uma aplicação. Além disso, esse requisito é essencial também para garantir a segurança da aplicação. Isso porque é essencial detectar o mais rápido possível anomalias, tanto na aplicação quanto em sua infraestrutura, que indiquem comportamentos anormais que podem estar relacionados à segurança, como acessos indevidos, ataques DDoS e vazamento de dados.

Existem diferentes técnicas e ferramentas que podem ser usadas para o monitoramento de aplicações relacionadas à segurança. Talvez a abordagem mais comum seja a utilização de sistemas de detecção e prevenção de intrusões (IDS/IPS - *Intrusion Detection System* e *Intrusion Prevention System*). Essas ferramentas monitoram o tráfego de rede em busca de padrões conhecidos de ataques; caso algum ataque seja detectado, elas podem alertar administradores do sistema ou até bloquear temporariamente o acesso aos recursos que podem estar sob ataque.

Auditoria

Saber quem e quando uma atividade foi executada é indispensável em várias aplicações. Isso pode ser importante tanto na aplicação diretamente quanto na infraestrutura dela. Um exemplo bem comum de aplicações que necessitam disso são as que possuem um fluxo que precisa de revisões e aprovações, nas quais é importante saber exatamente a hora que alguma ação foi feita, quem fez essa ação e quais os dados de entrada e saída dessa ação.

Um exemplo de aplicação que possui esse tipo de fluxo é a de assinaturas digitais, onde um usuário gera um contrato e o envia para as partes assinarem. Todas as partes devem assinar o documento de alguma forma, seja pela validação do e-mail ou telefone celular, e confirmar a assinatura. Todas as assinaturas

devem ser salvas em uma trilha que mostra exatamente quais pessoas assinaram e quando as assinaturas foram feitas. No próximo capítulo, veremos como esse tipo de trilha de auditoria pode ser implementada.

Além da auditoria que pode ser implementada em software, também é indispensável ter uma trilha de auditoria na infraestrutura da aplicação. No capítulo 8, foram mostrados os fluxos de CI/CD, que podem fazer a implantação da aplicação em um servidor. É indispensável saber quando e quem disparou esses fluxos, para saber quem contatar caso algum problema aconteça na implantação da aplicação. Além disso, todo sistema operacional salva o usuário que fez alguma modificação no servidor. No próximo capítulo, discutiremos várias boas práticas para ter uma auditoria confiável na infraestrutura da aplicação.

Vulnerabilidades

Um dos problemas mais complexos, e muitas vezes ignorados, em aplicações é o tratamento de *vulnerabilidades*. Essas vulnerabilidades normalmente são brechas na segurança da aplicação que possibilitam diversos tipos de ataques. Alguns exemplos de ataques são acessos indevidos a servidores, execução de código malicioso ou ataques para tornar uma aplicação indisponível.

Existem vários tipos de vulnerabilidades, que vão desde erros de programação, como falta de validação dos dados de entrada e uso de algoritmos criptográficos falhos, até erros na configuração da infraestrutura, como a utilização de senhas fracas. Além disso, muitas vezes não é dada a devida atenção à versão das dependências que são usadas na aplicação ou na infraestrutura. Além de problemas de compatibilidade, versões antigas podem ter várias brechas de segurança que podem ser usadas por usuários maliciosos. Essas vulnerabilidades podem estar diretamente na aplicação, como um erro de programação que permite a execução de um código malicioso, ou na infraestrutura, com o uso de uma

versão antiga de um sistema operacional em um servidor que tem uma falha de segurança conhecida que pode ser explorada.

Em uma aplicação pequena, é possível gerenciar manualmente essas vulnerabilidades, mas, com o crescimento de uma empresa ou de uma aplicação, essa tarefa fica praticamente impossível. Felizmente, existem diversos softwares de monitoramento que detectam possíveis vulnerabilidades na aplicação e sugerem mudanças para a correção das vulnerabilidades ou, pelo menos, a mitigação desses problemas. Um exemplo, que será discutido mais detalhadamente no próximo capítulo, é o DependaBot, que é uma automação disponível no GitHub que monitora e sugere atualizações para as dependências de uma aplicação.

9.4 Certificações de segurança

Para garantir que uma aplicação siga os requisitos de segurança definidos pelo mercado, existem diversas certificações que atestam que uma empresa segue boas práticas de segurança. Em software, algumas das certificações mais procuradas são o ISO 27001 e o SOC 2. Essas certificações visam garantir que uma organização estabeleça e siga diversas normas de segurança no gerenciamento das aplicações e seus dados.

As regras dessas certificações são bem abrangentes e estão em diferentes níveis. Em um primeiro nível, elas exigem verificações básicas sobre os funcionários da organização, como: que haja treinamentos básicos de segurança de todos os envolvidos no processo de desenvolvimento de software; que todos os computadores de trabalho tenham antivírus instalado; e que todos os usuários dos repositórios de código habilitem mecanismos de 2FA (*two-factor authentication*) em suas contas.

No nível da aplicação, as certificações podem exigir: que existam ferramentas de monitoramento de dependências; que a implantação das aplicações deva seguir um processo de CI/CD que tenha auditoria de quem fez a ação e quando ela foi executada; e, também, que todo código que vá para produção tenha passado por um processo de revisão de código.

No nível de infraestrutura, elas exigem que apenas usuários específicos possam acessar a infraestrutura de produção, que o acesso seja feito obrigatoriamente via HTTPS e que todos os bancos de dados tenham uma política de backup bem definida.

Essas certificações são opcionais, mas em alguns tipos de aplicações, é praticamente indispensável tê-las. Por exemplo, se sua empresa vende soluções para outras grandes corporações, como bancos, construtoras ou grandes varejistas, dependendo do tipo de dado que essas empresas terão que inserir em sua aplicação, certamente elas vão pedir garantias de que seus dados sejam armazenados seguindo boas práticas de segurança e que os serviços prestados não vão parar de funcionar por um ataque. Por isso, é bem comum que softwares que são vendidos como serviços tenham esse tipo de certificação.

Conclusões

A segurança de uma aplicação é, sem dúvida, um dos requisitos não funcionais mais importantes para as organizações. Isso porque a falta de segurança pode comprometer o funcionamento de toda a aplicação e também a credibilidade da organização toda. Existem diversos riscos que toda aplicação enfrenta, como o vazamento de dados, a invasão da infraestrutura da aplicação e ataques para deixar a aplicação indisponível.

Apesar de ser um requisito essencial, é, sem dúvida, também um dos mais ignorados. Isso acontece porque os problemas de segurança, diferentemente dos de escalabilidade ou confiabilidade, não acontecem no uso normal da aplicação. Enquanto não há

nenhum ataque à aplicação, tudo aparenta normalidade. Por isso, é essencial que haja um trabalho preventivo para evitar brechas de segurança na aplicação desde o começo da implementação. Também é extremamente necessário um monitoramento ativo e permanente tanto da aplicação quanto de sua infraestrutura.

Capítulo 10
Segurança no código

Praticamente qualquer aplicação precisa implementar algum nível de segurança, nem que seja um simples cadastro de usuário com uma página de login para fazer um controle mínimo de acesso. Obviamente que o nível de segurança necessário depende bastante do tipo de aplicação que será implementado. Por exemplo, em uma aplicação feita para usuários finais que tem apenas um fórum de perguntas e respostas sobre programação e que não coleta nenhum dado sensível do usuário, um controle de acesso básico, apenas para identificar um usuário, pode já ser o suficiente.

Mas existem aplicações bem mais críticas: uma aplicação de banco, por exemplo, que possui diversos dados críticos do usuário e que permite fazer várias operações financeiras em nome daquele usuário, precisa de um mecanismo de segurança muito mais complexo, como autenticação de duas etapas usando diferentes dispositivos. Além disso, esse tipo de sistema (financeiro) também é bastante visado para ataques, por isso, além dos mecanismos básicos de controle de acesso, ele também precisa de mecanismos como o uso de algoritmos criptográficos avançados, modelos de inteligência artificial que detectam operações atípicas e que podem bloquear o acesso de um usuário, entre muitos outros mecanismos mais complexos.

A segurança é, sem dúvida, um dos requisitos mais difíceis de serem atingidos satisfatoriamente no desenvolvimento de software, e muitas vezes é também o mais negligenciado. Isso porque, normalmente, até que um problema aconteça, como um acesso indevido ou um vazamento de dados, tudo parece estar funcionando bem. Por isso, é essencial usar mecanismos e ferramentas prontas e seguir boas práticas para desenvolver aplicações seguras, tanto para os usuários quanto para a própria empresa. Neste capítulo, veremos

algumas dessas práticas e ferramentas para implementar a segurança em diversas partes da aplicação, como para fazer o controle de acesso, garantir a confidencialidade dos dados e o monitoramento da aplicação.

10.1 Confidencialidade

A **confidencialidade** é o requisito que indica que informações sensíveis e restritas devem ser protegidas contra acessos não autorizados. A parte mais importante para implementar esse requisito é o **controle de acesso**, que visa garantir que apenas usuários autorizados possam acessar a aplicação e seus recursos. Além disso, temos que garantir que dados sensíveis como senhas e dados bancários não sejam visíveis em casos de acessos indevidos, e isso pode ser alcançado com o uso de algoritmos criptográficos que permitem que os dados sejam utilizados internamente na aplicação, mas que não possam ser usados caso alguém consiga acessá-los diretamente.

Controle de acesso

Praticamente qualquer aplicação precisa controlar quais usuários podem entrar na aplicação e quais recursos cada usuário pode acessar. O primeiro processo — verificar se o usuário pode entrar na aplicação — é chamado de **autenticação**, e ele também tem o objetivo de identificar quem está acessando a aplicação. Esse processo normalmente é feito por um dado que identifica o usuário, como o seu e-mail ou o CPF e uma senha.

O segundo processo — verificar se o usuário tem permissão de acessar um recurso — é chamado de **autorização**. Neste processo, é possível atribuir permissões individualmente para cada usuário, mas é comum, na maioria das aplicações, cada usuário ter um **papel** – por exemplo, *administrador* ou *operador* –, que é uma

forma de definir as mesmas permissões para um conjunto de usuários. Vamos conhecer com mais detalhes como funcionam esses dois processos e como podemos implementá-los, começando pela autenticação.

Autenticação

A implementação dos mecanismos de autenticação é um requisito fundamental em praticamente qualquer aplicação. O requisito em geral parece simples: no front-end, é necessário criar um formulário básico para criação do usuário e outro de login; já no back-end, precisamos definir como salvar as credenciais do usuário (nome de usuário e senha) no banco de dados e criar as rotas para o cadastro de um novo usuário e para a autenticação do usuário.

Entretanto, nem sempre é tão simples implementar esses mecanismos de uma forma correta e que não tenha brechas de segurança. Por isso, existem diversos padrões que podem ser usados para a implementação da autenticação, como o OAuth2 e o JWT. Também existem diversos frameworks, como o Spring Security, e até ferramentas prontas, como o Auth0 e o KeyCloak.

Até algum tempo atrás, a autenticação era mais simples, pois a maioria das aplicações armazenava uma sessão do usuário no servidor. Nesse modelo, um usuário fazia a autenticação no servidor e uma sessão era criada para esse usuário; o servidor enviava para o cliente um identificador dessa sessão, que normalmente era armazenado em um *cookie* no cliente, e então, em todas as requisições, o cliente enviava para o servidor o identificador da sessão. Quando o usuário fazia o *logout* da aplicação, bastava destruir a sessão do usuário no servidor. Esse mecanismo era todo implementado pelo servidor de aplicação, por isso pouco código era necessário para o controle de usuário.

No entanto, agora que a maioria das implementações de back-end usa uma API REST, que não mantém a sessão dos usuários, a implementação ficou mais complexa, pois normalmente a

implementação dos mecanismos de segurança deve ser feita diretamente na API ou em alguma ferramenta que trabalha junto com a API. Uma forma bastante comum de implementar isso é ter um serviço específico que cuida da autenticação, normalmente chamado de ***Identity and Access Manager*** (IAM).

Algumas funcionalidades básicas de um IAM são:

- **Armazenar dados de usuários:** o IAM deve ter uma rota para o cadastro de novos usuários, que deve armazenar no mínimo o identificador do usuário (nome de usuário, CPF etc.) e a sua senha. Podem existir dados opcionais como nome, endereço e telefone. Também devem existir rotas para alterar dados do usuário, redefinir a senha e excluir um usuário;
- **Rota de login:** deve existir uma rota para que o usuário seja autenticado na aplicação, normalmente verificando o identificador do usuário e sua senha, ou outros mecanismos, tais como certificados digitais. Essa rota normalmente devolve um token que deve ser usado para as próximas requisições da aplicação. Existem várias tecnologias para a geração e validação de tokens;
- **Verificação de token:** após o login, todas as requisições para um serviço devem enviar o token gerado, normalmente no cabeçalho da requisição. O IAM deve ter um serviço que verifica se esse token é válido e se ele ainda não expirou.

Nesse tipo de implementação, quando um usuário faz o login em uma aplicação, normalmente é retornado um token que o usuário deve enviar no cabeçalho nas requisições subsequentes para outras rotas da aplicação. Existem várias formas de implementar esse gerenciamento de token: a mais simples é gerar um token, armazená-lo em uma estrutura de dados da aplicação e, quando o usuário fizer uma requisição com o token, verificar se esse token existe.

O código a seguir mostra como pode ser implementada uma rota de login usando essa estratégia:

```
@PostMapping("/api/login")
public String getToken(@RequestBody LoginDTO loginDTO) {
    User user = userRepository
        .getUserByEmailAndPassword(
            loginDTO.getEmail(), loginDTO.getPassword());

    if (user != null) {
        String token = generateToken(user);
        return storeUserToken(token, user);
    }
    throw new UserNotFoundException();
}
```

No código, o primeiro passo é buscar o usuário usando o e-mail e a senha que foram enviados para a rota. Se o usuário for encontrado, é gerado um token, que pode ou não usar algum dado do usuário para ser gerado — normalmente é usado um algoritmo de geração de dados aleatórios. No fim, esse token é armazenado em uma estrutura de dados, como um `HashMap`, por exemplo, com o token como chave e as informações do usuário como valor. Caso o usuário não seja encontrado, é retornada uma exceção indicando que o login não funcionou.

Esse mecanismo funciona, mas tem um problema: cada usuário logado na aplicação terá um token armazenado. Para aplicações com poucos usuários, isso não é um problema, mas imagine um cenário de uma aplicação que pode ter milhões de usuários, serão necessário muitos recursos computacionais para armazenar todos esses tokens. Além disso, essa estratégia pode não ser muito segura, pois alguém que tenha acesso aos tokens consegue fazer o login na aplicação com os dados de outro usuário.

Para resolver esses problemas, existe um padrão chamado JWT (JSON Web Tokens – https://jwt.io/) para o gerenciamento de tokens. Um token JWT é dividido em três partes: a primeira é um

cabeçalho que possui o algoritmo usado para a assinatura do token; a segunda parte é um JSON que pode possuir dados do usuário, como o e-mail, e também a data de expiração do token; e a última parte, que é uma assinatura, pode ser usada para verificação da autenticidade do token e para confirmar que os dados do token não foram alterados pelo cliente.

Este é um exemplo do formato de um token JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InRlc3RlQHRlc3RlLmNvbSIsInJvbGUiOiJBRE1JTiIsImlhdCI6MTgyMjc3OTYzOH0.51fO9vWxan_hB43BNqChOccHwnOPA7rR5OnahaGdzVc
```

As três partes do token são separadas por um ponto, e todas elas são codificadas usando `Base64`. Os dados na primeira parte do token indicam o algoritmo e o tipo do token, que são:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

A segunda parte é formada por um JSON que pode possuir qualquer atributo que seja importante para a aplicação. No JWT, esses campos são chamados de `claims`. Alguns dados que normalmente são colocados em um JWT são o e-mail do usuário que está usando o token e o seu papel na aplicação. Dois dados bastante importantes que vão nessa parte é o `iat`, que é a data que o JWT foi gerado, e o `exp`, que é a data que o token vai expirar. Ambos os dados são representados em um `timestamp`.

```
{
  "email": "teste@teste.com",
  "role": "ADMIN",
  "iat": 1822779638
}
```

A terceira parte do token é a assinatura dos dados da primeira e segunda partes. Com essa assinatura, é possível verificar duas coisas

importantes: se os dados da primeira e segunda partes do JWT não foram modificados (um usuário poderia, por exemplo, tentar mudar o seu papel na aplicação); e validar se o JWT foi realmente gerado pela aplicação — um JWT que não tenha sua assinatura reconhecida será rejeitado pela aplicação.

O código a seguir faz algumas pequenas alterações no exemplo anterior para, em vez de usar um token gerado aleatoriamente, usar um gerador de JWT:

```
@PostMapping("/api/login")
public JWT getToken(@RequestBody LoginDTO loginDTO) {
    User user = userRepository
        .getUserByEmailAndPassword(
            loginDTO.getUsername(), loginDTO.getPassword());

    if (user != null) {
      return generateJwtToken(user);
    }
    throw new UserNotFoundException();
}
```

É possível gerar um token JWT manualmente, mas não é recomendado fazer isso. Atualmente, existem diversas bibliotecas para a geração de JWT em qualquer linguagem — no site oficial (https://jwt.io/libraries), há uma listagem de algumas bibliotecas que podem ser usadas para a geração e validação de JWTs.

Agora que temos o token gerado, podemos verificar se ele é válido ou não com o uso de um filtro, que será executado antes de todas as requisições para a API. Os filtros existem em qualquer tecnologia que implementa aplicações web. No Spring, é possível implementar um filtro usando a classe `OncePerRequestFilter`. Nessa classe, podemos interceptar a requisição e buscar todos os dados dela. Por exemplo, podemos usar o método `request.getHeader('Authorization')` para recuperar o token e verificar se ele é válido ou não. Esse método pode ser utilizado tanto para tokens gerados pela aplicação quanto para tokens JWT.

```
public class AuthorizationFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(
        HttpServletRequest request,
        HttpServletResponse response,
        FilterChain filterChain)
            throws ServletException, IOException {

        if (isUserAuthorized(request)) {
            filterChain.doFilter(request, response);
        } else {
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
        }

    }

    private boolean isUserAuthorized(
        HttpServletRequest request) {

        String token = request.getHeader('Authorization');
        // Aqui vai a lógica para verificar o token,
        // isso pode ser feito com um token
        // básico apenas com dados aleatórios ou
        // também com um token JWT
    }
}
```

Autorização

Agora que já vimos algumas possibilidades de implementar a autenticação, temos que conseguir verificar se o usuário possui **autorização** para acessar um recurso. Em uma API, entender o que é um recurso é simples: basicamente precisamos verificar se aquele usuário possui permissão para acessar uma determinada rota REST.

Existem várias formas de verificar se um usuário possui acesso em uma rota REST, e a mais simples é, logo no início dessa rota, buscar os dados do usuário, verificar o papel dele e se este papel é válido

nesta rota. O exemplo a seguir mostra uma das maneiras de fazer essa implementação:

```
@RestController
@RequestMapping("/api")
public class PessoasController {

    @GetMapping("/private")
    public String getPrivateResource(
            @RequestHeader(name="Authorization") String token) {

        User user = getUser(token);
        if (isUserAuthorized(user)) {
            return "Acesso autorizado.";
        } else {
            return "Você não tem acesso ao recurso.";
        }

    }

}
```

Nesse caso, para toda rota, seria necessário buscar os dados do usuário (ou apenas extrair os dados de um token JWT) e verificar se ele tem um dos papéis que são obrigatórios para acessar essa rota. Esse método funciona, mas tem um problema: haverá uma grande repetição desse código, já que em todas as rotas em que houver a obrigatoriedade de um papel para o acesso será obrigatório colocar esse `if` que faz a verificação dos papéis do usuário.

Para evitar a repetição de código em todas as rotas, uma possível solução é utilizar uma anotação que terá a função de verificar os papéis dos usuários antes de acessar o método. Assim fica bem mais fácil verificar a permissão do usuário. Essa é a estratégia usada pelo `spring-security`.

```
@RestController
public class MyController {
```

```
    @GetMapping("/private")
    @PreAuthorize("hasRole('USER')")
    public String getPrivateResource() {
        return "Acesso autorizado.";
    }
}
```

Nesse caso, toda rota que for exclusiva para um ou mais papéis precisa ser anotada com `@PreAuthorize`, e usuários que não tiverem essa permissão receberão uma resposta HTTP 403, que indica que a pessoa não tem permissão para acessar aquele recurso.

Eu descrevi os requisitos básicos para a implementação de autorização e autenticação em aplicações, mas como esse é um problema bastante comum, existem diversos padrões e protocolos para padronizar a implementação desses requisitos. Eu não vou entrar em detalhes sobre eles aqui, pois eles são bastante complexos, mas trarei uma breve descrição dos dois principais protocolos, que são o **OAuth2** e o **SAML2**.

OAuth2 (*Open Authorization 2.0*)

É um protocolo de autenticação e autorização bastante utilizado na internet para que aplicações permitam um fluxo de acesso por um usuário a recursos protegidos, sem que seja necessário compartilhar as credenciais de um usuário entre os serviços.

Nesse protocolo, normalmente existe um serviço de autenticação que possui os dados dos usuários e também os dados de serviços terceiros que podem usar a autenticação. Quando um usuário tenta usar um serviço terceiro sem estar autorizado, ele é direcionado para o serviço de autorização que verifica suas credenciais; depois, ele é redirecionado novamente para o serviço inicial, que confia na resposta do serviço de autenticação. Esse protocolo é muito usado para utilizar o *Social Login* (ou login social) de grandes empresas como Google, Facebook e GitHub para aplicações terceiras.

SAML2 (*Security Assertion Markup Language 2.0*)

É um protocolo de segurança baseado em XML e que possibilita a troca de dados de usuários de aplicações de forma segura. Ele define quais e em qual formato os dados devem ser trocados entre as aplicações. É o protocolo-base para a implementação do *Single Sign-On* (SSO), que é a tecnologia que permite que usuários façam login uma vez em um sistema e, em seguida, sejam autenticados automaticamente em outros sistemas.

Criptografia

Dados sensíveis devem ser armazenados ou transmitidos de uma forma que seja impossível saber o seu valor, ou seja, mesmo que alguém consiga acessá-los, não conseguirá saber o valor exato que está lá, pois os dados estão **criptografados**. Alguns exemplos de dados que podem ser criptografados são a senha de acesso ao sistema dos usuários e dados de pagamentos como cartão de crédito. Isso é bastante importante, pois alguns usuários devem ter acesso aos dados: um administrador de um banco de dados, por exemplo, talvez tenha que acessar a tabela de usuários de uma aplicação em alguma situação, mas ele não deve conseguir ler os dados que estão lá.

A **criptografia** consiste em cifrar um texto para transformar um dado legível em ilegível. Existem diversos tipos de criptografia em software; os mais conhecidos são a *simétrica*, a *assimétrica* e a *hash*:

- **Criptografia simétrica:** nesse método, uma única chave é usada tanto para criptografar quanto para descriptografar os dados. A mesma chave é compartilhada entre as partes envolvidas na comunicação. O desafio da criptografia simétrica é garantir que a chave secreta seja mantida segura, já que qualquer pessoa que tenha acesso a ela poderá descriptografar os dados;
- **Criptografia assimétrica (chave pública):** nesse método, um par de chaves é usado — uma chave pública e uma chave

privada. A chave pública é usada para criptografar os dados, enquanto a chave privada correspondente é usada para descriptografar. A principal vantagem desse método é que a chave pública pode ser amplamente compartilhada sem comprometer a segurança, pois apenas a chave privada correspondente pode descriptografar os dados criptografados com a chave pública;
- **Criptografia de *hash*:** são algoritmos que produzem, a partir de um dado de entrada, uma sequência única de caracteres. Esse tipo de criptografia é muito usado para o armazenamento de senhas em banco de dados, pois ele é irreversível, isto é, quando os dados são criptografados, é impossível descriptografá-los. Para a verificação de uma senha, quando um usuário digita a sua senha, o dado de entrada é novamente criptografado e, então, comparado com o que está armazenado no banco de dados.

Apesar de os algoritmos criptográficos serem bastante complexos, usá-los na maioria dos casos é bastante simples. Por exemplo, com o Spring, o algoritmo mais usado para criptografar as senhas dos usuários é o BCrypt, que é um algoritmo de criptografia de *hash*. Para usar esse algoritmo em uma aplicação Spring, basta utilizar a biblioteca `spring-security` e configurar o uso do algoritmo em uma classe de configuração, como mostra o código a seguir:

```
@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Depois, é possível utilizar essa classe `BCryptPasswordEncoder` em qualquer parte da aplicação, como no código a seguir:

```
@Autowired
BCryptPasswordEncoder bCryptPasswordEncoder;

@GetMapping("/encrypt")
public String encrypt(@RequestParam("texto") String texto) {
```

```
    return bCryptPasswordEncoder.encode(texto);
}
```

Além do Spring, existem diversas outras bibliotecas que disponibilizam algoritmos como o BCrypt em praticamente qualquer linguagem de programação.

> **MD5**
>
> Talvez o algoritmo criptográfico mais conhecido seja o MD5, que foi muito usado nas aplicações mais antigas por ser extremamente fácil de usar. Porém, atualmente seu uso não é recomendado, pois ele é um algoritmo vulnerável a ataques de força bruta.

10.2 Monitoramento na segurança

Já falamos de monitoramento no capítulo sobre confiabilidade, mas esse requisito é também bastante importante para a segurança da aplicação. O monitoramento é feito principalmente por ferramentas que analisam os logs e a infraestrutura da aplicação, porém também são necessárias algumas ações diretamente na aplicação. Um exemplo é a geração de logs de auditoria das ações que os usuários executam na aplicação, o que permite identificar usuários que realizam possíveis ações fraudulentas ou identificar possíveis tentativas de acesso indevido ou de invasões à aplicação. É necessário sempre analisar o código-fonte da aplicação para tentar identificar erros no código ou bibliotecas que possam conter brechas de segurança.

Auditoria

Permitir a auditoria em uma aplicação é um requisito bastante importante em vários tipos de sistema. Em uma aplicação em que é

necessário saber qual usuário tomou alguma ação, é indispensável ter um registro indicando exatamente qual a ação tomada, qual o horário dela e, se possível, os dados de entrada e saída da operação (sempre tomando cuidado para não salvar dados sensíveis). Esse requisito é chamado normalmente de **trilha de auditoria**. Um exemplo bem comum para os desenvolvedores são os *commits* em um repositório de código: eles podem ser pensados como uma auditoria que serve para saber exatamente o que cada usuário fez e em qual momento.

A trilha de auditoria pode ter várias finalidades. A primeira e mais óbvia é ter o registro de tudo o que aconteceu na aplicação, mas, além disso, com ela é possível também entender como um determinado sistema chegou a um certo estado e conseguir desfazer ações indesejáveis. Ela também serve como um backup e um repositório de dados, que seriam perdidos caso esse histórico não fosse salvo em nenhum outro lugar. O `json` a seguir mostra um exemplo de como esses dados podem ser salvos em uma aplicação:

```
{
    "usuario":  "usuario@sistema.com",
    "acao": "SALVAR_USUARIO",
    "data": "2023-07-05T22:14",
    "dadosEntrada": {
        "email": "novo@sistema.com",
        "senha": "******",
    }
}
```

Nesse exemplo, temos o usuário, representado por seu *e-mail*; a ação que foi feita, *salvar um novo usuário*; a *data* em que o evento aconteceu; e os *dados de entrada* na chamada dessa ação. Note que a senha está escondida: isso é uma boa prática para evitar que dados sensíveis sejam salvos no log de auditoria. Os dados salvos em uma trilha de auditoria na aplicação podem variar bastante, mas o objetivo é sempre ter um log de todas as transações que foram feitas na aplicação.

A análise desses logs pode também levar a ações para a segurança da aplicação. Por exemplo, um usuário que está tendo uma sequência de ações incomuns ou que realize uma ação que ele não deveria ter permissão (por falha do sistema ou por algum ataque) pode ter o seu acesso bloqueado.

Vulnerabilidades

O objetivo do monitoramento de vulnerabilidades é detectar problemas nas dependências das aplicações. Toda aplicação usa um grande número de dependências, e essas dependências também possuem outras dependências, o que forma uma grande árvore que pode chegar a dezenas ou até centenas de bibliotecas. É impossível monitorar manualmente se alguma vulnerabilidade foi encontrada em alguma dessas bibliotecas. Para isso, existem diversas ferramentas que buscam ativamente bibliotecas em que houve comunicação de vulnerabilidades.

Um exemplo de ferramenta desse tipo é o DependaBot, que verifica periodicamente as dependências de uma aplicação. Essa ferramenta pode ser configurada em um repositório do GitHub e funciona para vários gerenciadores de dependências, como o Maven e o Gradle para Java, o PIP para o Python e o RubyGems para Ruby. Por exemplo, para configurar o DependaBot em uma aplicação Java que usa o Maven, basta adicionar um arquivo no path `/.github/dependabot.yml` com a seguinte configuração:

```
version: 2
updates:
  - package-ecosystem: "maven"
    directory: "/"
    schedule:
      interval: "weekly"
```

Com essa configuração, o DependaBot vai verificar semanalmente o arquivo `pom.xml`, que possui todas as dependências da aplicação.

Caso ele encontre alguma dependência desatualizada, ele abre automaticamente um *pull request* com a atualização da biblioteca. Ele verifica ainda qual a compatibilidade da versão antiga com a nova (nem sempre esse valor está disponível). Se essa atualização tiver a resolução de algum problema crítico de segurança, o DependaBot ainda cria um alerta que indica qual é o problema que está sendo corrigido. A figura a seguir mostra uma lista com diversos alertas criados pelo DependaBot:

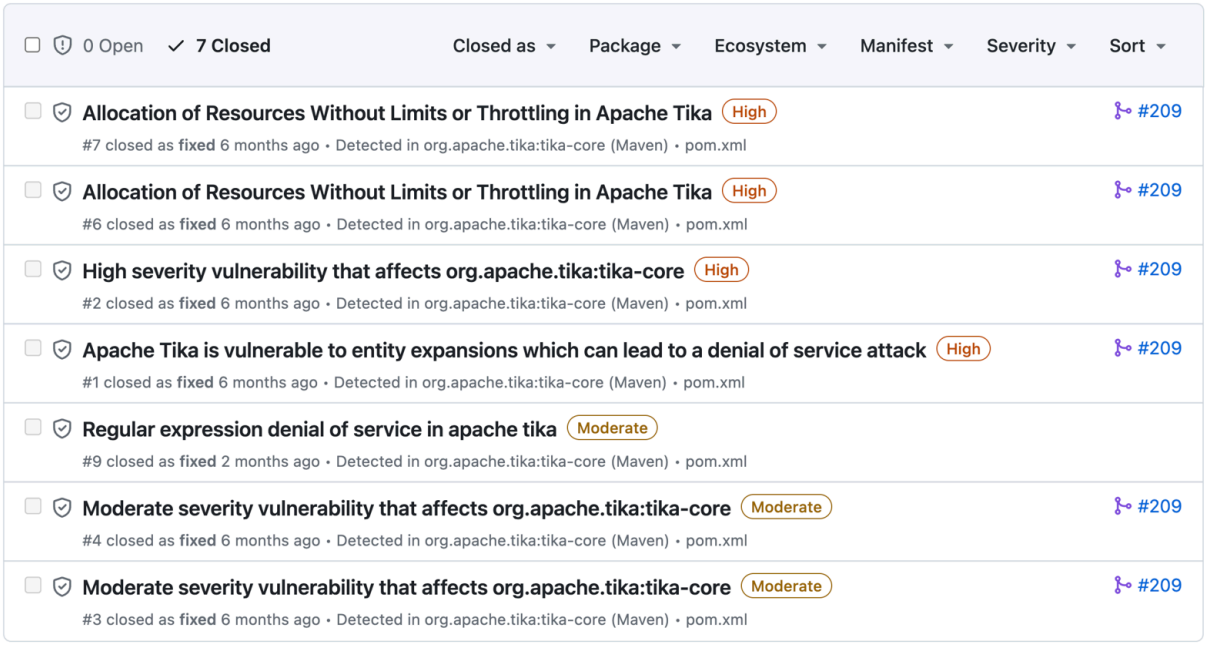


Figura 10.1: Lista de vulnerabilidades nas dependências

As vulnerabilidades de segurança mais críticas são catalogadas e tornadas públicas por um grupo na internet, o CVE (https://www.cve.org/), o que facilita bastante o trabalho de verificar as dependências que podem ter algum problema. Um dos pontos mais importantes definidos pelo CVE é o nível de criticidade da vulnerabilidade. Por exemplo, existem vulnerabilidades que são chamadas de `0-day vulnerability`, que indicam que elas devem ser resolvidas em algumas horas — com elas, suas aplicações estão extremamente expostas a ataques que são fáceis de fazer. O quadro a seguir relata uma vulnerabilidade *0-day* que ficou bastante famosa, o Log4Shell.

LOG4SHELL

Uma vulnerabilidade desse tipo que ficou bastante famosa recentemente (em dezembro de 2021) foi o problema encontrado no Log4J, uma biblioteca bastante utilizada para a geração de logs em aplicação Java.

O Log4J permite que variáveis ou expressões sejam incluídas nas mensagens de logs, mas uma falha nesse recurso permitia que códigos maliciosos fossem incluídos na mensagem de logs, e esses códigos eram executados no servidor. Esse código malicioso podia ser mandado em um campo de entrada enviado para o servidor, e se os dados desse campo eram adicionados no log, a falha acontecia quando o Log4J fazia a escrita do log.

Essa vulnerabilidade ficou conhecida como Log4Shell, e foi catalogada com o código CVE-2021-44228.

10.3 Outros problemas de segurança

Vimos alguns dos principais problemas de segurança que podem acontecer em uma aplicação, mas obviamente há diversos outros possíveis ataques aos quais as aplicações podem estar vulneráveis e com os quais pessoas desenvolvedoras e arquitetas de software precisam lidar. Nesta seção, vamos discutir outros três desses problemas, o SQL Injection, o CSRF (*Cross-Site Request Forgery*) e o XSS (*Cross-Site Scripting*), que são alguns dos ataques mais comuns aos quais aplicações web podem estar vulneráveis.

Existem muitos outros tipos de ataque, e um bom lugar para consultá-los é o site da OWASP, a *Open Worldwide Application Security Project* (https://owasp.org/), uma organização que tem o objetivo de melhorar a segurança de software.

SQL Injection

O **SQL Injection** talvez seja a vulnerabilidade mais comum em aplicações web. A ideia é que a pessoa que ataca a aplicação consegue inserir (ou "injetar") um código SQL malicioso em uma aplicação que acessa um banco de dados. Esse ataque visa explorar falhas na validação dos dados de entrada de uma aplicação para executar comandos SQL em um banco de dados. Isso pode permitir que o atacante acesse, modifique ou exclua dados e até mesmo execute operações de administração no banco de dados, como a exclusão de tabelas ou até mesmo de um banco de dados inteiro.

Geralmente, a injeção de SQL ocorre quando os dados de entrada fornecidos por um usuário da aplicação não são validados adequadamente antes de concatená-los em consultas SQL. Os atacantes exploram essa falta de validação ao inserir caracteres especiais ou sequências de escape que enganam o sistema e alteram o significado das consultas SQL executadas.

Por exemplo, analise o comando SQL a seguir para a listagem de produtos de uma categoria:

```
SELECT nome, descricao, preco
FROM produtos
WHERE categoria = {categoria};
```

Esse comando SQL espera um parâmetro para ser executado: o identificador da categoria dos produtos que serão buscados. Se o parâmetro for passado corretamente, nada de errado será executado. Porém, um usuário malicioso pode tentar mandar para a categoria o seguinte valor:

```
' ; DROP TABLE produtos; --
```

O `--` é o caractere de comentário no SQL e é usado para comentar o que vier depois do comando injetado. O comando executado será

o seguinte:

```
SELECT nome, descricao, preco
FROM produtos
WHERE categoria = ''; DROP TABLE produtos; --';
```

Por ser um problema já bastante conhecido, já existem várias formas de proteger a aplicação desse ataque. As principais são validar e sanitizar todos os dados de entrada de uma aplicação e usar frameworks ORM (*Object Relational Mapping*) como o Hibernate, que já fazem a validação para evitar SQL Injection.

Cross-Site Request Forgery (CSRF)

O CSRF (*Cross-Site Request Forgery*, ou falsificação de solicitação entre sites, em uma tradução livre) é um tipo de ataque bastante comum em aplicações na internet. Nele, uma página maliciosa pode se passar por uma página real e, usando credenciais válidas do sistema real armazenadas em *cookies*, tentar fazer requisições para esse sistema real enganando um usuário. Um exemplo desse tipo de ataque é uma página que imita um formulário de um sistema real com um formulário qualquer, mas que, em vez de chamar o serviço que está no formulário, chama um serviço para trocar a senha do usuário sem que a pessoa saiba.

O principal mecanismo para evitar esse ataque é o CORS (*Cross-Origin Resource Sharing*), que é um mecanismo de segurança implementado pelos navegadores web para verificar se o servidor aceita a origem da requisição. Quando o browser vai fazer uma chamada para um serviço na web, antes de enviar a requisição em si (como uma GET, POST, PUT), ele envia uma requisição com o verbo `HTTP OPTIONS` para o servidor, que responde as seguintes informações:

- `Access-Control-Allow-Origin` : as origens que são aceitas pelo serviço;

- `Access-Control-Allow-Methods` : os métodos HTTP que aquele serviço aceita;
- `Access-Control-Allow-Headers` : os cabeçalhos que aquele serviço aceita.

O browser então verifica as condições da requisição (origem, método HTTP e cabeçalhos); se todas estiverem de acordo com o que o servidor respondeu, ele faz a requisição para o servidor. Caso contrário, a requisição será bloqueada, evitando o que poderia ser um ataque do tipo CSRF. É importante ressaltar que essa verificação é feita pelo browser, requisições feitas de outros clientes ou de ferramentas como o Postman não serão bloqueadas.

Essa configuração pode ser feita em qualquer serviço web. No Spring, por exemplo, essa configuração pode ser feita da seguinte forma:

```
@Configuration
@EnableWebMvc
public class CorsConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
            .allowedOrigins("https://meusite.com")
            .allowedMethods("GET", "POST", "PUT");
    }
}
```

Nesse exemplo, apenas requisições que sejam feitas a partir de `https://meusite.com` e com os métodos `GET`, `POST` e `PUT` serão permitidas. É possível liberar o CORS para qualquer origem também. Para isso, basta colocar no método `allowedOrigins` o valor `*`, mas não é recomendável, já que sua aplicação aceitará requisições de qualquer origem.

Cross-Site Scripting (XSS)

O XSS (*Cross-Site Scripting*) é um ataque em que usuários tentam injetar um script malicioso (geralmente JavaScript) em páginas web vistas por outros usuários. Isso pode acontecer em sites que possuem campos de entrada que depois serão exibidos para outros usuários, como fóruns e bate-papos. Esse ataque funciona em serviços que não validam e não sanitizam os dados de entrada enviados por usuários na internet.

O fluxo de um ataque XSS normalmente é:

- **Injeção de código:** um usuário mal-intencionado insere em um campo de entrada de uma aplicação web um código JavaScript malicioso;
- **Exibição em página web:** os dados inseridos pelo usuário mal-intencionado são armazenados em um banco de dados sem que nenhuma validação seja feita. Um usuário legítimo entra na aplicação e visualiza os dados de entrada que foram inseridos pelo usuário malicioso;
- **Execução no navegador:** o navegador do usuário legítimo interpreta o código JavaScript malicioso e o executa. Isso pode resultar em ações indesejadas, como roubo de *cookies* de autenticação, redirecionamento para páginas maliciosas ou exibição de *pop-ups* de *phishing*.

Para prevenir o XSS, um sistema deve principalmente validar as entradas dos usuários, e isso pode ser feito tanto no front-end quanto no back-end de uma aplicação. Além disso, outras ações podem evitar esse ataque, como sempre escapar a saída em páginas web, para evitar que o código HTML seja interpretado pelo browser, e utilizar bibliotecas de sanitização de dados de entrada. Um exemplo é a `OWASP Java HTML Sanitizer`, que permite que usuários usem HTML na entrada de dados, mas identifica tentativas de ataques XSS e invalida a entrada do usuário.

Conclusões

Garantir a segurança de uma aplicação é, sem dúvida, um dos requisitos mais difíceis de ser implementado. Existe uma infinidade de possíveis ataques e vulnerabilidades aos quais as aplicações estão sujeitas. Por isso, é indispensável implementar diversos mecanismos diretamente no código-fonte da aplicação para garantir a sua segurança.

Sem dúvida, o requisito mais importante relacionado à segurança é o controle de acesso para permitir que apenas usuários *autenticados* e *autorizados* (guarde bem essas palavras e tenha a diferença entre elas bem clara) tenham acesso aos recursos de uma aplicação. Além disso, é indispensável fazer o monitoramento de tudo que está acontecendo na aplicação para que casos suspeitos e usos fora do padrão da aplicação possam ser detectados e corrigidos caso seja necessário.

Outra coisa bastante importante é conhecer os principais ataques que usuários mal-intencionados tentam fazer em aplicações na internet. Aqui, citamos alguns exemplos como o SQL Injection, o CSRF e XSS, mas existem diversos outros ataques aos quais aplicações podem estar vulneráveis. É essencial que pessoas arquitetas de software estejam sempre se atualizando sobre os ataques mais comuns e sobre técnicas para evitar esses tipos de ataques.

Já falamos bastante sobre segurança, mas ainda estamos apenas na metade deste tópico. Agora que o nosso código já está seguro, temos que garantir que a infraestrutura da aplicação também esteja protegida com múltiplas camadas de segurança e também possua vários mecanismos de detecção de tentativas de ataque, mas isso é assunto para o próximo capítulo.

Capítulo 11
Segurança na infraestrutura

Produzir um código seguro é extremamente importante para evitar brechas na segurança das aplicações. Porém, tão importante quanto é garantir a segurança da infraestrutura e dos recursos que são usados pela aplicação. Por exemplo, apenas pessoas autorizadas devem ter acesso às máquinas que executam uma aplicação, ao seu banco de dados e aos seus arquivos de configuração, e pouquíssimas pessoas devem ter acesso a esses recursos; além disso, todos os acessos devem ser logados e monitorados.

Neste capítulo, veremos que alguns dos mecanismos de segurança que são usados no código da aplicação também devem ser usados na infraestrutura, como o controle de acesso, por exemplo — em qualquer sistema, controlar o acesso aos recursos de uma aplicação é extremamente importante para garantir a sua segurança.

Também devemos definir diversas maneiras para evitar ataques externos em aplicações que estão disponíveis na internet. Por exemplo, na maioria dos casos o servidor que está executando uma aplicação não deve estar exposto diretamente na internet, mas sim usar alguma tecnologia que redireciona as requisições para o servidor certo, como balanceador de carga e API Gateways. Assim, é possível limitar bastante o acesso direto à máquina e adicionar camadas de proteção e controle de acesso, que seria bem mais difícil de implementar se o acesso ao servidor fosse direto.

Muitos dos erros que deixam a infraestrutura de uma aplicação vulnerável são feitos por descuido ou por falta de monitoramento dos recursos. Por isso, é essencial utilizar também ferramentas que analisam a infraestrutura de uma aplicação continuamente. Alguns exemplos incluem verificar se alguma senha ou segredo está sendo usado diretamente no código-fonte que está disponível no sistema de controle de versão e quais portas estão abertas em um servidor,

pois apenas as portas usadas diretamente pelas aplicações devem estar abertas para acesso externo.

Vamos começar a analisar a segurança da infraestrutura de uma aplicação considerando o mesmo problema do capítulo anterior, a **confidencialidade**, para evitar acessos indevidos a qualquer parte da aplicação.

11.1 Confidencialidade na infraestrutura

Alguns dos problemas que devem ser evitados na infraestrutura são os mesmos que os da aplicação. Por exemplo: não devemos permitir que usuários sem autorização acessem qualquer recurso que uma aplicação use, tais como servidores, bancos de dados, código-fonte, ambientes de nuvem ou arquivos de configuração. Isso também é feito com controle de acesso usando autorização e autenticação; a diferença é que, neste caso, normalmente usamos ferramentas já prontas para monitorar a infraestrutura e não implementamos um controle de acesso inteiro.

Também devemos monitorar todos os acessos para evitar acessos indevidos e para conseguir rastrear possíveis problemas nas permissões dos usuários. A criptografia aqui é essencial também: nunca devem ser armazenadas em textos abertos senhas que as aplicações usam ou qualquer outro dado que contenha informações sensíveis. Além disso, essas senhas de configuração devem ser armazenadas em lugares seguros, como cofres de senhas, e nunca em lugares aos quais várias pessoas possuem acesso, como o código-fonte da aplicação.

Controle de acesso

O controle de acesso na infraestrutura da aplicação é um grande desafio, pois podem existir dezenas, ou até centenas, de

ferramentas, servidores, repositórios, bancos de dados, entre muitas outras coisas para as quais o controle de acesso tem que ser feito. Isso varia muito de acordo com o nível de maturidade da empresa. Em empresas que ainda não têm uma maturidade muito grande na infraestrutura, os acessos costumam ser bagunçados e normalmente são dados de acordo com a necessidade — se uma equipe precisa de acesso ao servidor de produção, logo é dado o acesso para facilitar a resolução de um problema. Isso pode até ajudar na agilidade na solução de um problema, mas é uma prática que pode deixar as aplicações vulneráveis.

Já em empresas em que o nível de maturidade é maior, normalmente o acesso direto à infraestrutura é dado a pouquíssimas pessoas. Habitualmente, quase todo acesso à infraestrutura é feito via ferramentas que fazem um controle de acesso bastante limitado com base no tipo de usuário, e apenas usuários com altíssimos níveis de privilégio conseguem acessar a infraestrutura diretamente. Por exemplo, na grande maioria das empresas maiores, o acesso a um servidor de produção é dado apenas a usuários que tenham um nível de privilégio muito alto, e todo acesso deve ser logado. Usuários como desenvolvedores que precisam, por exemplo, acessar o log de uma aplicação devem fazer uma requisição pedindo o arquivo de log de uma aplicação de um determinado período.

Em ambientes de nuvem como a AWS e o GCP, é possível ainda separar os ambientes de produção e teste em contas ou subcontas separadas, impedindo qualquer acesso de usuário que não possua permissão em cada um dos ambientes. Gerenciar o controle de acesso em todas as ferramentas e recursos de uma empresa é bastante difícil, por isso existem diversas políticas e controles que devem ser seguidos para garantir que tudo esteja adequado. Vamos discutir alguns deles.

O **Princípio do Menor Privilégio (*Least Privilege*)** define que os usuários devem receber apenas as mínimas permissões necessárias para realizar as suas tarefas, e esses acessos devem ser concedidos sempre sob demanda; ou seja, usuários não devem receber vários

privilégios logo que iniciam suas tarefas, mas, sim, quando for necessário. Um exemplo onde isso acontece é nos repositórios de código da empresa: uma nova pessoa desenvolvedora não deve ter acesso irrestrito a todos os repositórios imediatamente, mas receber as permissões sempre que for trabalhar em um projeto específico.

É essencial em ferramentas e recursos que são bastante críticos exigir uma **autenticação forte** dos usuários, exigindo uma senha robusta que contenha letras, números e caracteres especiais. Além disso, deve-se utilizar a autenticação de dois fatores (2FA) em aplicações que permitem seu uso e que são bastante críticas, como em repositórios de código, ambientes de nuvem e e-mails. Um exemplo de 2FA é a utilização de códigos que são enviados via e-mail ou SMS e que o usuário deve inserir na hora da autenticação. Por fim, utilizar serviços que permitam o acesso com o uso de chaves, como o acesso via SSH para servidores.

Com o número grande de ferramentas que são necessárias gerenciar, é bastante importante sempre **bloquear contas inativas** para evitar acessos de usuários que não possuem mais autorização na aplicação. Uma forma de resolver esse problema é usar um serviço que centraliza as informações dos usuários em uma empresa, como um AD (*Active Directory*), com o qual é possível ativar e inativar um usuário ou até o seu acesso para uma ou mais aplicações em apenas um lugar.

As máquinas dos usuários estão conectadas à rede da empresa, seja em um ambiente presencial ou até mesmo em uma VPN, se o usuário estiver em um ambiente remoto. As máquinas de usuários são bem mais vulneráveis que os servidores que executam as aplicações, pois usuários podem acessar sites maliciosos, abrir e-mails infectados, ou até inserir pen drives com vírus nas máquinas. Por isso, muitas empresas fazem a **segregação de redes**, separando as redes de produção das aplicações das redes de usuários ou até mesmo das redes dos ambientes de teste.

Outro item que é bastante importante e que é cobrado em qualquer certificação de segurança é o **treinamento dos usuários** que operam a infraestrutura da aplicação. A maioria das empresas que trabalham com dados de usuários são obrigadas a prover treinamentos a seus funcionários sobre diversos temas, como segurança da informação, ataques comuns como *phishing*, e leis específicas como a LGPD.

Como vimos, controlar o acesso a toda a infraestrutura de execução das aplicações em uma empresa pode ser um grande desafio, e é bastante comum haver falhas nesse processo. Por isso, é essencial usar as políticas aqui descritas e, também, ferramentas que facilitem o gerenciamento dos usuários.

Gerenciamento de segredos

Praticamente toda aplicação precisa de um grande número de dados confidenciais para a sua execução. Alguns exemplos são a senha do banco de dados, as chaves para acesso de outras APIs, chaves de acesso a ferramentas de um provedor de nuvem (como o S3 da Amazon ou o Cloud Storage do GCP), entre muitas outras possibilidades. Um erro muito comum é deixar essas senhas ou chaves diretamente no código, o que é uma falha grave de segurança, pois elas ficam acessíveis para qualquer pessoa que tenha acesso ao repositório. Além disso, normalmente temos diferentes senhas para cada um dos ambientes, o que dificulta o gerenciamento das senhas se elas forem definidas diretamente no código.

Uma forma de resolver esse problema é utilizar um cofre de senhas, que é um sistema que as armazena de forma segura e controla o acesso aos segredos que serão usados nas aplicações. Nesse tipo de ferramenta, normalmente administradores do sistema podem cadastrar novos segredos (*secrets*), e qualquer usuário que precise usar esse segredo pode usá-lo sem ter acesso diretamente ao valor do segredo. Nessas aplicações, os segredos são normalmente armazenados em um formato de chave/valor. Por exemplo, um

administrador pode definir uma chave chamada `SENHA_BANCO_DE_DADOS` e colocar como valor a senha `senha_super_secreta`. Qualquer usuário que tenha acesso àquele cofre de senhas vai saber que existe a chave `SENHA_BANCO_DE_DADOS`, e poderá usar esse valor em diversos lugares, como em *pipelines* de CI/CD ou como variáveis de ambiente em um servidor.

Outro ponto importante é que normalmente os segredos nunca podem ser acessados via interface gráfica do cofre de senhas, nem mesmo pelos administradores da ferramenta que inseriram aquele segredo. Os segredos só poderão ser acessados via API para serem usados nas ferramentas que precisam deles. Os dados podem ser atualizados, mas sempre por valores novos.

Existem várias ferramentas de cofre de senhas, e uma das mais conhecidas é o **Vault**, que é uma ferramenta que pode ser usada gratuitamente se gerenciada diretamente pelo usuário (existe um serviço na nuvem já pronto para uso, mas ele é pago). Para quem usa o GitHub, outra opção é o Secrets, que também é uma ferramenta de gerenciamento de segredos que pode ser usada no GitHub Actions. A imagem a seguir mostra uma tela do **GitHub Secrets**, que tem dois segredos registrados, `AWS_ACCESS_KEY_ID` e `AWS_SECRET_ACCESS_KEY`, que são as chaves para acessar a AWS em uma aplicação.

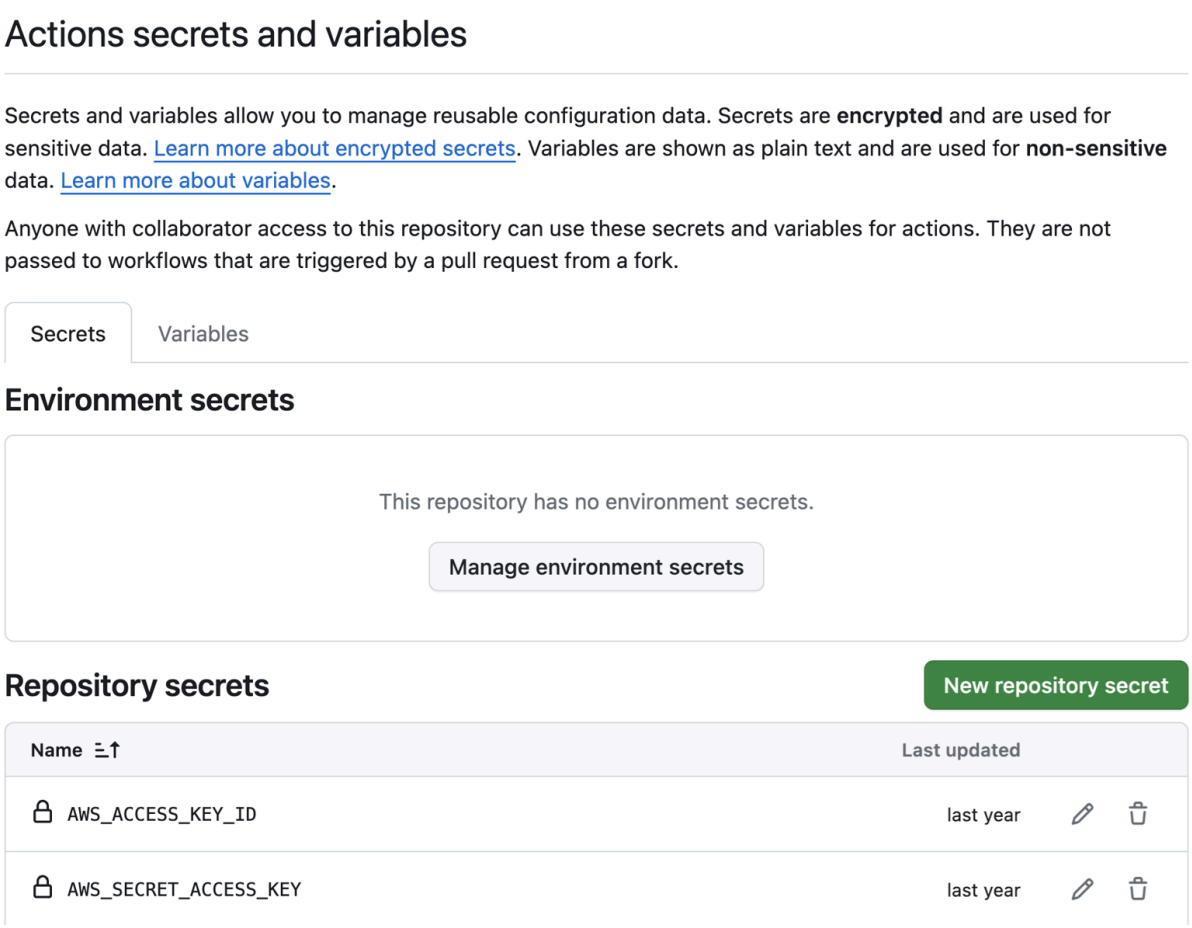


Figura 11.1: Exemplos de segredos no GitHub

Com esses segredos registrados, agora é possível usar os valores registrados neles diretamente em um *pipeline* de CI/CD, como o que vimos no capítulo 8, *Confiabilidade na infraestrutura*. Veja como podemos usar o valor de um segredo na listagem a seguir:

```
- name: Deploy to EC2 instance
  run: |
    ssh -o StrictHostKeyChecking=no -i ${{ secrets.EC2_PRIVATE_KEY }} ec-
2user@ip-address
    docker run -d -e SENHA_BANCO_DE_DADOS=${{ secrets.SENHA_BANCO_DE_DADOS
}} app
```

Pegamos a última parte do arquivo que criamos no capítulo 8 e fizemos duas modificações. Lá, a chave SSH estava colocada diretamente no código e, como vimos anteriormente, isso é uma

falha grande de segurança, por isso mudamos para um segredo chamado `EC2_PRIVATE_KEY`. Além disso, estamos passando para a aplicação a senha do banco de dados que ela deve acessar usando o segredo `SENHA_BANCO_DE_DADOS`.

11.2 Monitoramento

Já falamos bastante de monitoramento em capítulos anteriores, mas esse requisito é também essencial para garantir a segurança da aplicação. Qualquer serviço que está exposto na internet é suscetível a sofrer tentativas de ataque. Essas tentativas podem vir de qualquer lugar do mundo e muitas vezes são feitas por robôs que tentam encontrar vulnerabilidades em máquinas expostas na internet.

O monitoramento deve ser feito em várias partes da infraestrutura da aplicação. Alguns exemplos são:

- **Logs do sistema operacional:** O log do sistema operacional pode conter diversos registros sobre tentativas de acessos não autorizados ou sobre usuários que tentam executar comandos ou acessar arquivos sem permissão;
- **Logs de servidores e aplicações:** os servidores de aplicações normalmente possuem diversos tipos de logs, como logs de acessos e das aplicações. Com eles, é possível detectar tentativas de acesso fora do padrão esperado e tentativas de acesso não autorizadas;
- **Tráfego de rede:** o tráfego de rede pode mostrar tentativas de intrusão na aplicação, de envio de arquivos maliciosos, de ataques de força bruta que tentam quebrar senhas, e o escaneamento de portas abertas;

- **Tráfego de e-mail:** diversos ataques são feitos por e-mail para infectar computadores que estão conectados em uma rede, e é possível monitorar o tráfego de e-mail em busca de *malware*, *phishing* e *spam*;
- **Vulnerabilidades conhecidas:** assim como existem ferramentas para monitorar as dependências nas aplicações, também existem ferramentas que monitoram a infraestrutura em busca de vulnerabilidades, como serviços do sistema operacional que podem ter vulnerabilidades e versões de *firmware* desatualizadas.

A quantidade de itens para monitorar é enorme, por isso é impossível fazer isso manualmente. Atualmente, existe um grande número de ferramentas que podem ser utilizadas e que monitoram diversos itens da infraestrutura da aplicação. Por exemplo, a Amazon AWS possui um serviço chamado GuardDuty, que monitora todo o ambiente de uma empresa, analisando logs, tráfego de rede e acessos, e emite alertas sempre que algum problema é encontrado. O GuardDuty pode emitir alertas sobre robôs que ficam escaneando as portas abertas em um servidor para tentar encontrar falhas conhecidas em aplicações que rodam naquelas portas.

Dependências e patches

Assim como é necessário monitorar e atualizar as dependências na aplicação, é também necessário verificar a atualização de servidores e aplicações que fazem parte da infraestrutura ou que são usados pela aplicação. Se as aplicações foram desenvolvidas em Java, de tempos em tempos é necessário atualizar a versão da máquina virtual dessa linguagem. Apesar de parecer simples atualizar pacotes, esse tipo de atualização pode ser complicado, já que uma versão diferente de um pacote ou de uma ferramenta (como a máquina virtual do Java) pode ter diversos impactos no funcionamento de uma aplicação.

Por isso, é comum muitas empresas demorarem para fazer esse tipo de atualização. O Java mesmo é um bom exemplo: apesar de a linguagem e a plataforma já estarem na versão 21, muitas empresas ainda usam versões antigas da plataforma, especialmente a versão 8, que foi bastante popular. Isso faz com que desenvolvedores não tenham muitas das novas funcionalidades da linguagem e não tenham várias melhorias que foram feitas na plataforma, e podem até ter situações de risco de vulnerabilidade caso a versão antiga não receba mais atualizações de segurança (especificamente com o Java isso não é problema, pois mesmo a versão 8 ainda está recebendo atualizações de segurança).

Os sistemas operacionais normalmente avisam quando há aplicações ou pacotes que estão desatualizados. Por exemplo, em um acesso SSH em uma máquina Linux no AWS, a primeira mensagem exibida é a de que existem pacotes desatualizados, como mostra a figura a seguir:

```
Last login: Thu Sep 14 19:19:04 2023 from 177.34.169.128

       __|  __|_  )
       _|  (     /   Amazon Linux 2 AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-2/
149 package(s) needed for security, out of 212 available
Run "sudo yum update" to apply all updates.
```

Figura 11.2: Mensagem que indica que existem pacotes desatualizados no servidor

Isso ajuda na atualização dos pacotes, mas se a aplicação for muito grande, é praticamente impossível entrar em todas as máquinas e recursos de uma aplicação e verificar se há atualizações pendentes. Por isso, existem várias ferramentas disponíveis que podem facilitar essa verificação. Na AWS, existe o **AWS Systems Manager**, que permite a instalação e atualização de pacotes em diversas máquinas de uma só vez.

11.3 Auditoria

Uma má prática que é muito comum em diversas empresas é não utilizar usuários específicos para acessar a infraestrutura. Por exemplo, toda máquina no EC2 da Amazon possui um usuário padrão chamado `ec2-user`, e muitas empresas simplesmente usam sempre esse usuário para fazer qualquer ação diretamente nas máquinas. Isso é uma grande falha de segurança, pois não permite que uma **auditoria** completa seja feita na infraestrutura da aplicação, sendo impossível identificar qual usuário fez uma ação específica em algum recurso da aplicação.

A auditoria é um item obrigatório em todos os processos de certificações de segurança, como o ISO 27001 e o SOC 2. Por isso, esse requisito deve ser seguido em todas as ferramentas que são utilizadas pela empresa. Alguns exemplos de onde a auditoria deve ser feita são:

- **Código-fonte das aplicações:** é essencial saber por quem e quando um código foi adicionado em uma aplicação. Isso é importante para garantir a qualidade e a segurança das aplicações e para saber o que pode ter causado um problema, seja ele de segurança ou não;
- **Acessos a servidores:** as máquinas onde as aplicações são executadas possuem diversas informações confidenciais, e por elas passam todas as informações que são processadas pelas aplicações. Então todos os acessos e todas as ações executadas nessas máquinas devem ser logados;
- **Tráfego na rede:** todas as informações de uma empresa passam pela rede dela, por isso é indispensável o monitoramento contínuo de todos os acessos que são feitos na rede. Todos os usuários devem ser identificados e deve-se evitar ao máximo o acesso anônimo em máquinas conectadas à rede da empresa;

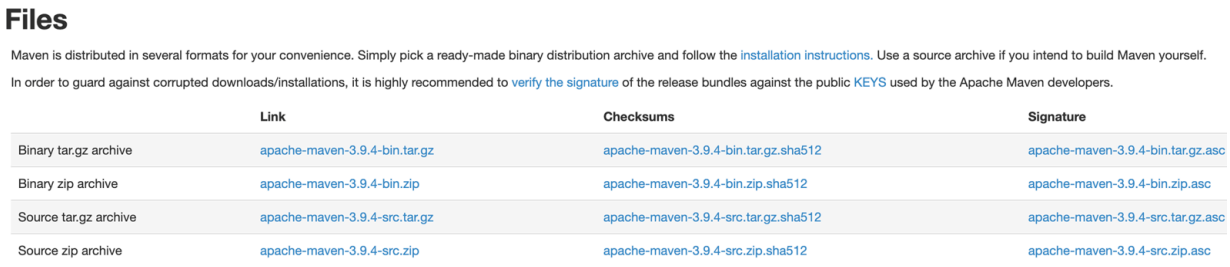
- **Execução de fluxos de CI/CD:** como vimos no capítulo *Confiabilidade na infraestrutura*, a maioria das implantações e atualizações das aplicações atualmente é feita de forma automatizada usando um fluxo de CI/CD. Por isso, é essencial saber por quem e quando esses fluxos foram inicializados, mesmo que eles sejam inicializados automaticamente por alguma mudança no código;
- **Ambientes de nuvem:** grande parte das aplicações é executada em ambientes de nuvem atualmente, então o acesso e todas as ações executadas nesses ambientes devem ser logados, para que seja possível analisar e identificar acessos e ações que estão fora do padrão esperado.

11.4 Integridade

A **integridade** é o requisito de confirmar que algo não foi comprometido por alguma operação ilegal ou não autorizada, e esse termo pode ser usado para várias partes de uma aplicação. Podemos dizer que os dados de uma aplicação em um banco de dados estão íntegros se tivermos a certeza de que não houve nenhuma transação inválida que os alterou ou inseriu dados que não são válidos na aplicação. Também podemos verificar a *integridade da infraestrutura* da aplicação; para isso, é preciso validar que todas as máquinas estão executando exatamente o que deveriam, que nenhuma foi acessada indevidamente e teve seu comportamento alterado.

Um exemplo de onde a verificação de integridade é bastante usada é no download de arquivos: em muitos lugares, quando vamos baixar um arquivo, é mostrado um *hash* que foi calculado desse arquivo. Esse *hash* é calculado usando algoritmos de *hash* como o BCrypt, comentado no capítulo anterior, e deve ter sempre o mesmo valor para um mesmo arquivo. Então, quando fazemos um download de um arquivo, para verificar se foi baixado realmente o mesmo

arquivo que estava no servidor, podemos verificar novamente o *hash* em nossa máquina e comparar com o *hash* informado pelo servidor. A imagem a seguir mostra a página de download do Apache Maven. Nela, além dos links para o arquivo, há uma coluna chamada `checksums` : esse dado é o *hash* criado por algum algoritmo de *hash*.



Files

Maven is distributed in several formats for your convenience. Simply pick a ready-made binary distribution archive and follow the installation instructions. Use a source archive if you intend to build Maven yourself.

In order to guard against corrupted downloads/installations, it is highly recommended to verify the signature of the release bundles against the public KEYS used by the Apache Maven developers.

	Link	Checksums	Signature
Binary tar.gz archive	apache-maven-3.9.4-bin.tar.gz	apache-maven-3.9.4-bin.tar.gz.sha512	apache-maven-3.9.4-bin.tar.gz.asc
Binary zip archive	apache-maven-3.9.4-bin.zip	apache-maven-3.9.4-bin.zip.sha512	apache-maven-3.9.4-bin.zip.asc
Source tar.gz archive	apache-maven-3.9.4-src.tar.gz	apache-maven-3.9.4-src.tar.gz.sha512	apache-maven-3.9.4-src.tar.gz.asc
Source zip archive	apache-maven-3.9.4-src.zip	apache-maven-3.9.4-src.zip.sha512	apache-maven-3.9.4-src.zip.asc

Figura 11.3: Página de download do Apache Maven com o campo 'checksums'

Essa verificação de *hash* serve tanto para segurança — pois podemos verificar o *hash* para validar se o arquivo que estava descrito no servidor é realmente o que foi baixado — quanto para a confiabilidade — pois podemos validar também se o arquivo está sendo baixado corretamente. Essa estratégia do *hash* é usada em vários outros tipos de troca de dados, como imagens do Docker e dados enviados para repositório de arquivos como o AWS S3.

A validação da integridade de um servidor não é tão simples de ser feita como a de um arquivo, mas também é importante. Para validar a integridade do hardware da aplicação, são usadas várias técnicas. A principal é o **monitoramento contínuo dos servidores**, e a partir desse monitoramento são buscados eventos que estão fora do padrão normal da aplicação. Se uma aplicação tem uma média de mil requisições por minuto, mas de repente acontece um grande pico de acesso com cem vezes mais requisições que o normal, talvez o servidor esteja recebendo um ataque de DDoS (*Distributed Denial of Service*). Esse tipo de monitoramento pode ser feito para outras partes da infraestrutura de uma aplicação, como o banco de dados, por exemplo.

11.5 Tipos de ataques

Se um recurso estiver disponível na internet, é quase certo que ele vai sofrer tentativas de ataque em algum momento. Vamos passar por alguns dos principais tipos de ataques que podem acontecer e aprender a tentar identificá-los para nos protegermos.

Talvez os ataques mais conhecidos que existem são o ataque de **negação de serviço** *(DoS – Denial of Service)* e o **ataque distribuído de negação de serviço** *(DDoS – Distributed Denial of Service)*. Os dois têm o mesmo objetivo: neles, os atacantes realizam uma grande quantidade de requisições para um serviço, gerando um volume de tráfego muito grande, talvez muito maior do que a aplicação consegue suportar, o que a faz sair do ar. A melhor forma de evitar esse tipo de ataque é usar um *firewall* que detecta máquinas maliciosas que podem estar fazendo um ataque DDoS e filtra as requisições delas. O AWS, por exemplo, possui um serviço chamado AWS Shield que tem especificamente a responsabilidade de evitar esse tipo de ataque.

O **ataque de força bruta** é um tipo de ataque que pode acontecer tanto na infraestrutura quanto na própria aplicação. Nele, os atacantes tentam descobrir as senhas dos usuários em alguma parte da aplicação testando todas as possibilidades de senha que podem existir na aplicação. Esse tipo de ataque funciona em sistemas que usam senhas fracas — por exemplo, senhas pequenas e que possuem apenas caracteres de um tipo — e que não possuem um mecanismo de autenticação de dois fatores. A infraestrutura de uma aplicação pode estar vulnerável a esse tipo de ataque, pois existem muitas ferramentas e aplicações que podem ser invadidas, como servidores, ambientes de nuvem, banco de dados e repositórios de código.

Outro ataque bem conhecido é o **escaneamento de porta**, no qual invasores escaneiam todas as portas de um servidor para tentar encontrar portas abertas e, depois, fazer outro ataque, que é

explorar vulnerabilidades nas aplicações que estão executando naquela porta. Por exemplo, o SSH normalmente é executado na porta 22 de um servidor; um servidor web, na porta 80; e o PostgreSQL usa como padrão a porta 5432. Um atacante pode identificar que essas portas estão abertas em um servidor e tentar explorar alguma vulnerabilidade conhecida nas aplicações que rodam nessas portas. O ideal para evitar esse tipo de ataque é evitar que muitas portas estejam abertas para qualquer IP na internet e manter as aplicações sempre atualizadas para que não haja falhas de segurança que possam ser exploradas. Uma medida paliativa que pode ajudar nesse caso é não usar as portas padrão nas aplicações, o que pelo menos evita ataques automáticos.

Muitas vezes um usuário pode ter um acesso legítimo a um recurso, mas ele terá um papel específico que pode não ter muitos privilégios. Por exemplo, às vezes um usuário tem acesso a um servidor Linux, mas ele não tem o privilégio de *super usuário* (usando o comando `sudo`, por exemplo), o que limita que esse usuário realize várias ações no servidor. Porém, existe um ataque chamado de **escalada de privilégio**, em que um usuário, legítimo ou não, tenta aumentar seu privilégio para poder realizar qualquer ação no sistema. A forma de evitar esse tipo de ataque é manter os níveis de privilégio o mais baixos possível para todos os usuários e monitorar tentativas de ações ilegais.

Um último ataque que também está se tornando bastante comum e já atingiu algumas grandes empresas nos últimos anos é o ***ransomware***, no qual uma pessoa criminosa, ao conseguir acessar a infraestrutura de uma aplicação, criptografa todos os dados que estão nos servidores, impedindo que a vítima desse ataque acesse os dados. Depois de um tempo, os atacantes exigem da vítima o pagamento de um resgate para que os dados sejam descriptografados e os acessos sejam restabelecidos. Um *ransomware* é feito normalmente quando um atacante consegue acesso a uma máquina dentro da rede de uma empresa. Algumas formas de evitar ou mitigar esse tipo de ataque incluem: manter os

privilégios dos usuários da rede o mais baixos possível; manter segregadas as redes internas e de execução das aplicações; e manter sempre atualizados softwares e dependências. Além disso, é importante sempre manter backup dos dados, para que, caso um ataque desse tipo ocorra, seja possível restabelecer os serviços o mais rápido possível.

Conclusões

Manter a infraestrutura de uma aplicação protegida de ataques é uma atividade extremamente complexa, principalmente pela grande quantidade de recursos que precisam ser configurados e monitorados. Qualquer aplicação, por mais simples que ela seja, terá pelo menos um servidor, um banco de dados e um sistema de controle de versão de código. Portanto, é essencial utilizar ferramentas e padrões para garantir que a aplicação esteja o mais segura possível, além de seguir boas práticas para evitar problemas como vazamento de dados e acessos indevidos.

Com este capítulo, concluímos a discussão sobre os três requisitos não funcionais centrais quando pensamos na arquitetura de uma aplicação. Obviamente, existem outros vários requisitos não funcionais que são extremamente importantes e que a pessoa arquiteta de software também deve conhecer bem para o desenvolvimento de uma aplicação, porém eles não são tão centrais quando pensamos na execução da aplicação. Alguns desses requisitos são a *usabilidade*, a *manutenibilidade* e a *portabilidade*. No próximo capítulo, discutiremos esses e alguns outros requisitos.

Parte 5: Outros requisitos não funcionais

Capítulo 12
Outros requisitos

Este livro focou nos três requisitos não funcionais mais importantes na hora da definição da arquitetura de uma aplicação e na sua execução, que são o *desempenho*, a *confiabilidade* e a *segurança*. Mas existem diversos outros requisitos que também devem ser levados em consideração no desenvolvimento de uma aplicação para que ela atenda plenamente à expectativa dos seus usuários.

Neste capítulo, discutiremos alguns desses outros requisitos, como:

- A **usabilidade**, com o objetivo de avaliar o quão fácil é operar a aplicação;
- A **portabilidade**, para verificar se a aplicação está preparada para ser executada em diferentes condições e ambientes;
- A **corretude** e a **testabilidade**, que indicam se as funcionalidades da aplicação estão implementadas de forma correta e quão fácil é testar e verificar essas funcionalidades;
- A **manutenibilidade**, que avalia o nível de dificuldade para manutenção e evolução da aplicação;
- A **reprodutibilidade**, que possibilita que os resultados obtidos na aplicação sejam reproduzidos de forma fácil;
- A **documentação**, para avaliar se a aplicação está bem documentada e descrita tanto para usuários quanto para desenvolvedores;
- Por fim, o **custo** e o **tempo**, que vão afetar todos os outros requisitos, já que a maioria das decisões no desenvolvimento de software são tomadas por restrições financeiras e de tempo de mercado.

12.1 Usabilidade

A **usabilidade** é o requisito que define quão fácil e eficiente é a interação dos usuários com uma aplicação. Ela se concentra na experiência do usuário durante a utilização do software, buscando tornar a interação intuitiva, agradável e sem complicações. Uma boa usabilidade é fundamental para garantir a satisfação e a retenção dos usuários. A chance de um usuário abandonar uma aplicação é muito maior se a sua usabilidade for ruim, mesmo que ela atenda a uma demanda real desse usuário.

Existem diversos aspectos que precisam ser pensados na usabilidade de uma aplicação, tais como:

- **Interface do usuário (UI):** a interface do usuário é a forma pela qual os usuários interagem com o software. A interface de uma aplicação deve ser intuitiva, organizada e visualmente agradável. Elementos como menus, botões, campos de entrada, ícones e layout são projetados para serem claros e consistentes. Existem diversas bibliotecas que disponibilizam padrões e componentes que facilitam bastante essa tarefa, como o MUI (https://mui.com/);
- **Navegação:** refere-se à facilidade de movimentação dentro do software para acessar diferentes partes da aplicação. A navegação pode ser feita por meio de menus, links, guias ou outros elementos. É importante que os usuários consigam encontrar facilmente as funcionalidades desejadas e percorrer o software de maneira lógica e fluida;
- **Feedback:** é importante fornecer um feedback claro e imediato aos usuários para cada ação que ele realize na aplicação. Isso pode ser feito por meio de mensagens de sucesso, confirmação, alertas ou avisos de erros, tanto devido a uma ação incorreta do usuário quanto por um evento inesperado na aplicação. O feedback adequado ajuda os usuários a entender o que está acontecendo no sistema e corrigir possíveis problemas;

- **Documentação e ajuda:** a disponibilidade de documentação clara e ajuda contextual dentro do software contribui para a usabilidade, especialmente em aplicações complexas, nas quais dificilmente o usuário conseguirá realizar as ações intuitivamente. Instruções claras, tutoriais, documentação on-line e recursos de suporte ajudam os usuários a entender melhor o software, aprender a utilizá-lo e resolver problemas quando necessário.

A usabilidade também se conecta bastante com os requisitos não funcionais discutidos anteriormente. Por exemplo, o desempenho de uma aplicação afeta bastante a usabilidade, pois uma aplicação lenta e que demora para disponibilizar um feedback para as ações provavelmente levará a uma experiência ruim para o usuário. A confiabilidade também é importante, pois se o usuário se depara constantemente com erros na aplicação, a usabilidade da aplicação consequentemente também será ruim.

Apesar de a usabilidade ser um aspecto extremamente importante no desenvolvimento de software, não vamos nos aprofundar nesse requisito, pois ele normalmente não é uma função primordial da pessoa arquiteta de software (apesar de, obviamente, essa pessoa poder ajudar). A usabilidade é normalmente responsabilidade da equipe de definição do produto ou dos responsáveis pela experiência do usuário (UX), e a pessoa arquiteta de software deve ser capaz de proporcionar o ambiente para que a usabilidade planejada seja viável tecnicamente.

12.2 Portabilidade

A **portabilidade** refere-se à facilidade de mudança de uma aplicação de um ambiente para outro. Isso envolve diversas decisões, como a linguagem de programação, o banco de dados e as bibliotecas utilizadas. Além disso, com a utilização cada vez mais

comum de nuvens, também é preciso pensar na portabilidade de um provedor de nuvem para outro. Não são todas as aplicações que necessitam se preocupar com portabilidade, mas sem dúvida isso pode ser bastante importante em cenários de incerteza, nos quais pode ser necessário, no futuro, executar a aplicação em diferentes ambientes.

Um termo bastante conhecido que é relacionado à portabilidade de soluções é o *vendor lock-in*, ou, em português, "ficar preso em um fornecedor". Isso pode acontecer, por exemplo, em um banco de dados: todo banco de dados relacional usa a linguagem SQL para a definição da estrutura e para a manipulação de dados, mas todos os fornecedores possuem ferramentas e comandos específicos, que só funcionam naquele banco de dados. Se uma aplicação usa muitos desses recursos, a portabilidade da aplicação fica comprometida, pois mudar de banco de dados pode se tornar uma tarefa praticamente impossível.

Outro caso em que o *vendor lock-in* também está sendo bastante discutido atualmente é em ambientes de computação em nuvem. Ambientes como a AWS e o GCP fornecem um grande número de ferramentas para facilitar o desenvolvimento das aplicações. Porém, quanto mais dependente a aplicação é dessas ferramentas, mais a portabilidade da aplicação fica comprometida. Por isso, se a portabilidade é um requisito importante para a sua aplicação, você deve se preocupar em abstrair os detalhes dessas ferramentas para conseguir mudar o ambiente em que ela executa sem ter que reescrever todo o seu código.

Um exemplo são as ferramentas de sistema de arquivos, como o S3 da AWS. Se sua aplicação usa essa biblioteca da Amazon para acessar o S3, caso seja necessário executar a aplicação no GCP, será preciso reescrever o código dessa parte da aplicação. Nesse caso, uma possível solução seria usar a biblioteca `jclouds`, que possui uma camada de abstração sobre os principais fornecedores de alguns serviços na nuvem, permitindo a troca de provedor sem a necessidade de reescrever código.

12.3 Corretude

Talvez o mais importante dos requisitos seja garantir que a aplicação realize exatamente as funções para as quais ela foi planejada. Essa propriedade é chamada de **corretude**, e ela é verificada por testes funcionais que devem ser realizados pelas equipes que fazem as especificações dos requisitos funcionais de uma aplicação. Além disso, a corretude pode ser verificada também em nível de código; por exemplo, um teste de unidade ajuda a garantir que uma função específica produza sempre um resultado válido para uma determinada entrada.

Esse é um requisito essencial em qualquer aplicação, porém, em algumas, isso é ainda mais indispensável. Por exemplo, imagine uma aplicação de um fórum na internet que tem o objetivo de mostrar as mensagens mais recentes primeiro, mas, por um erro de implementação, ordena pelas mensagens mais antigas. Isso é um erro, mas não é nada muito grave e não gerará efeitos adversos difíceis de detectar ou corrigir. Porém, existem aplicações muito mais críticas em que um simples erro pode acarretar em diversos efeitos adversos. Um exemplo crítico são aplicações de banco: se uma transação for processada incorretamente, isso pode levar a erros que serão difíceis de serem detectados. Outro exemplo, ainda mais crítico, é um erro em um software de um avião, o que pode levar a graves acidentes.

Por isso, apesar de toda aplicação precisar de um processo bem feito na verificação da sua qualidade, existem aplicações que são muito mais críticas e que precisam garantir a sua corretude em todas as partes do sistema. Em aplicações normais, como aplicações web em geral, a corretude normalmente é verificada usando uma grande quantidade de testes, sejam eles manuais ou automatizados. Os testes possuem diversos níveis, começando com os testes de unidade, feitos diretamente no código, até os testes de interface, no qual robôs simulam o comportamento do usuário em uma aplicação (falaremos mais sobre testes na próxima seção).

Já aplicações mais críticas podem usar um conceito chamado **verificação formal de software**, que usa técnicas formais como lógica matemática, álgebra booleana e teoria dos autômatos para analisar o código ou o modelo do programa e provar matematicamente que ele se comporta conforme o esperado. Esse tipo de verificação é feito em diversos tipos de sistemas, como sistemas de controle de aeronaves e de tráfego aéreo, sistemas médicos, de veículos autônomos e de energia nuclear.

12.4 Testabilidade

Outro requisito não funcional bastante relacionado à *corretude* é a **testabilidade**, que indica a facilidade de testar uma aplicação. Existem várias decisões no desenvolvimento de software que podem facilitar ou dificultar os testes.

Esse requisito está diretamente ligado à qualidade da aplicação, já que, quanto mais fácil for testar, mais fácil será identificar e corrigir defeitos na aplicação. Existem diversos tipos de teste, entre eles: o **teste de unidade**, que normalmente é o teste feito diretamente no código-fonte; o **teste de integração**, que verifica se os diversos componentes de uma aplicação estão funcionando corretamente quando conectados; e o **teste de interface**, que testa a aplicação simulando a interação de um usuário com as interfaces da aplicação.

O objetivo da testabilidade é facilitar tanto a realização de testes manuais da aplicação quanto o desenvolvimento de testes automatizados. Algumas decisões que afetam a testabilidade de uma aplicação são:

- **Modularização:** quanto mais *modularizada* a aplicação, mais fácil é testá-la. Por exemplo, é bem mais fácil testar um monólito que foi bem dividido em camadas do que um monólito em que o código não está bem organizado. Também é mais fácil

testar métodos pequenos, que possuem apenas um objetivo, do que testar um método que possui várias responsabilidades;
- **Isolamento de dependências:** existem vários níveis de teste. O de mais baixo nível, que é o teste de unidade, deve testar apenas as funções desenvolvidas para a aplicação e evitar qualquer tipo de dependência. Por exemplo, para testar um código que acessa o banco de dados e faz algum cálculo com os dados, normalmente não queremos testar a conexão com o banco, já que isso é implementado por alguma biblioteca. O objetivo é testar apenas o cálculo isoladamente. Para isso, normalmente usamos os *mocks*, que são objetos que simulam os objetos reais, mas que podemos programar exatamente o que eles devem fazer;

- **Reprodução:** a grande maioria dos métodos de uma aplicação deve ser determinística, isto é, para uma determinada entrada, sempre produzir a mesma saída. Isso facilita bastante a realização de testes, já que uma das etapas dos testes automatizados é verificar se a resposta gerada é a mesma esperada. Testar algoritmos não determinísticos é um desafio bem mais complexo, já que fica muito mais difícil verificar se o resultado gerado é o esperado;

- **Cobertura:** é importante saber quais partes da aplicação estão cobertas por testes e quais não estão. Praticamente toda linguagem possui um framework de testes, como `pytest` para o Python e o `JUnit` para o Java. Todas elas possuem suporte nativo para a verificação de cobertura ou algum plugin que faz essa verificação. Além disso, essas ferramentas também podem gerar relatórios que indicam quais partes do código estão com coberturas satisfatórias de testes e quais não.

12.5 Manutenibilidade

Uma das características mais importantes de uma aplicação é se ela é fácil de ser mantida e evoluída. É bastante provável que uma aplicação não estará pronta em sua primeira versão e precisará continuamente de atualizações e correções de falhas durante o seu uso. O requisito não funcional que indica a facilidade de manutenção de uma aplicação é a **manutenibilidade**, e existem diversas ferramentas e métricas que podem ajudar a medi-la. Por exemplo, o **SonarQube** (https://www.sonarsource.com/products/sonarqube) é uma ferramenta que faz análise estática de código para encontrar possíveis *dívidas técnicas*, como mostra a figura a seguir. Esta imagem mostra problemas no código de uma aplicação que devem ser resolvidos para que ele siga boas práticas.

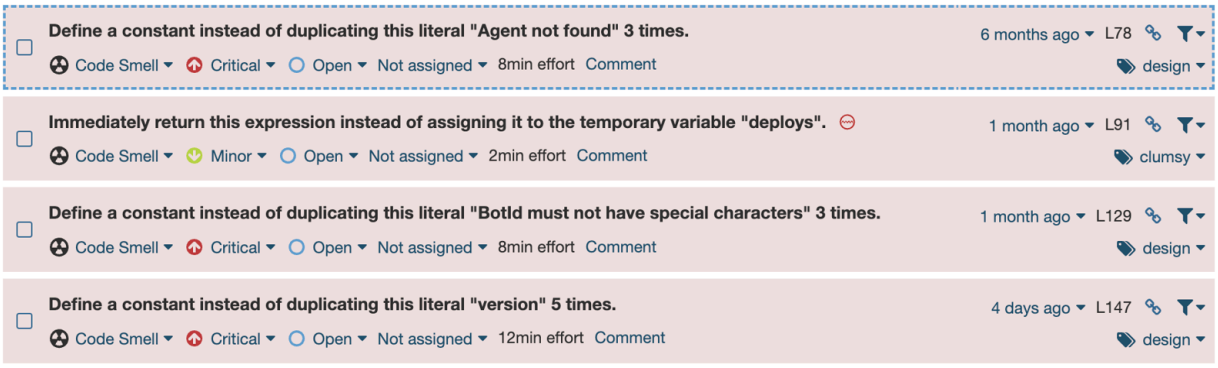


Figura 12.1: Problemas de manutenibilidade no SonarQube

Outra visualização também interessante do SonarQube é um gráfico que relaciona todas as classes de uma aplicação e mostra a cobertura de código (eixo Y) e uma estimativa das horas necessárias para corrigir todos as dívidas técnicas que existem em cada uma das classes (eixo X). A figura a seguir mostra um exemplo desse gráfico:

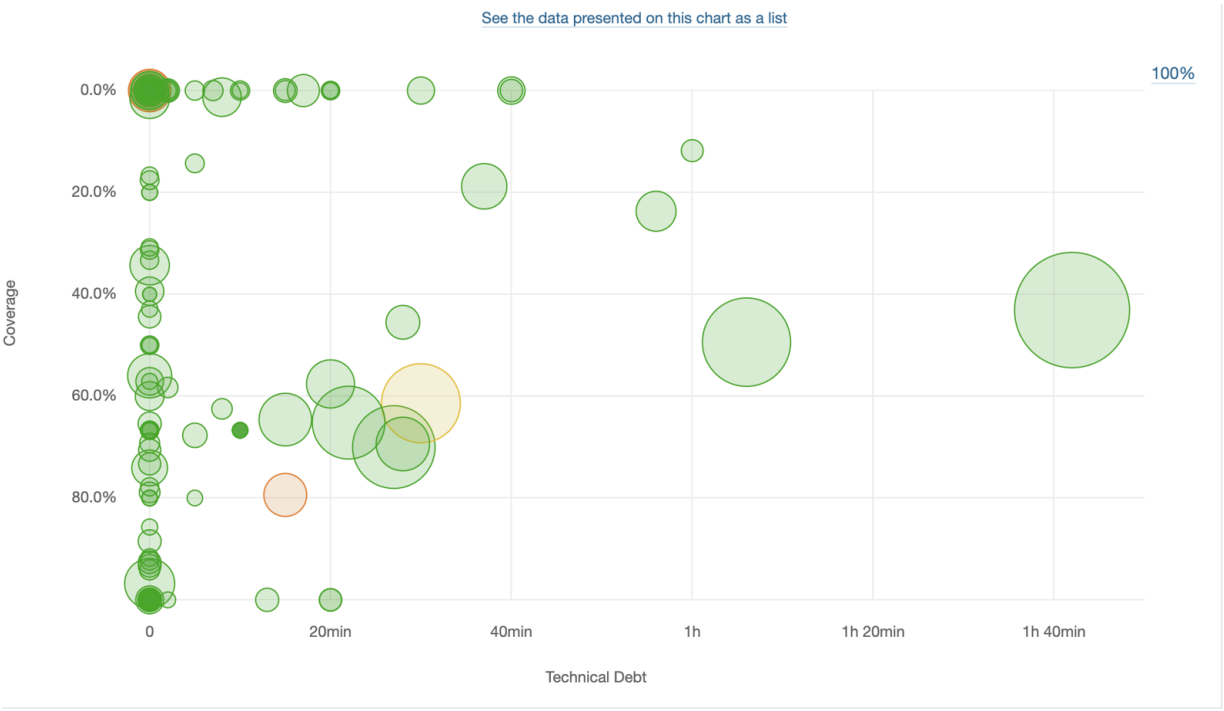


Figura 12.2: Cobertura de testes e problemas de manutenibilidade em classes da aplicação

DÍVIDA TÉCNICA

Muitas vezes precisamos entregar um requisito o mais rápido possível, mas para fazer o código mais correto possível seria necessário um grande tempo de desenvolvimento. Por isso, nesses casos podemos fazer uma escolha consciente de que a solução não será a melhor possível, visando rapidez na entrega. Apesar dessa escolha, podemos documentar esse problema para refatorar esse código posteriormente para a solução correta. Isso é chamado de **dívida técnica**. Esse termo é usado porque essa solução não muito boa é vista como uma dívida que deve ser paga o mais rápido possível, para que o problema não se torne ainda mais grave no futuro.

12.6 Reprodutibilidade

Qualquer aplicação é executada sempre em diferentes ambientes, como máquinas de desenvolvedores, servidores instalados em *data centers* de clientes e ambientes de nuvem. Mesmo uma aplicação que está sempre executando em um mesmo lugar pode ter variações no seu ambiente, como variações na carga de usuários e na carga da rede de comunicação. A capacidade da aplicação de gerar sempre as mesmas respostas e comportamentos consistentes, mesmo em diferentes condições, é chamada de **reprodutibilidade**. Esse requisito é importante em diversos fatores, como:

- **Testes de software:** os testes de software dependem bastante da reprodutibilidade, pois se um dos resultados ou comportamentos de uma aplicação não forem reprodutíveis, será praticamente impossível validar se um teste (principalmente os automatizados) funcionou ou não;
- **Auditoria:** em ambientes regulamentados, como o ambiente financeiro, por exemplo, muitas vezes é necessário verificar se os dados estão consistentes com o que era esperado, e a reprodutibilidade é necessária para cumprir requisitos de auditoria e conformidade;
- **Desenvolvimento colaborativo:** a maioria das aplicações é desenvolvida por times de desenvolvimento; além disso, em aplicações grandes, é normal que existam vários times que colaboram no desenvolvimento. A reprodutibilidade facilita bastante a colaboração, pois garante que os resultados sejam consistentes em diferentes ambientes de desenvolvimento.

Apesar de ser um requisito importante em grande parte das aplicações, existem vários softwares que não atendem a esse requisito. Um tipo de aplicação que não o segue são os LLMs (*Large Language Models*), que, de propósito, podem produzir resultados diferentes para uma mesma entrada. Isso é bastante comum em

aplicações que usam algoritmos probabilísticos e que possuem variabilidade aleatória.

12.7 Documentação

Muitos autores também consideram a **documentação** de um sistema um requisito não funcional. Existem diferentes tipos de documentação que são construídos para diferentes públicos.

Inicialmente, é necessário criar a documentação interna da aplicação, que descreve suas principais funcionalidades, sua arquitetura geral e contém outras informações importantes para o desenvolvimento e a manutenção da aplicação.

Outro tipo de documentação que é bastante importante é a documentação que explica as funcionalidades da aplicação para os usuários. Para sistemas muito simples, talvez essa documentação não seja necessária, mas para aplicações que são muito complexas e que dificilmente o usuário conseguiria usar sem uma grande ajuda, essa documentação é essencial. Um exemplo desse tipo de documentação são os manuais da AWS que descrevem cada uma das funcionalidades que ela disponibiliza. Essa documentação pode ser acessada em https://docs.aws.amazon.com/. Existem várias ferramentas e frameworks que podem ajudar no desenvolvimento de documentação para os usuários, como o MkDocs (https://www.mkdocs.org/) ou o GitHub Pages (https://pages.github.com/).

Além desses dois tipos de documentação, dependendo do projeto, podem existir outros documentos importantes também, como a documentação de segurança, que explica como os requisitos de

segurança são tratados descrevendo protocolos, algoritmos criptográficos e políticas de segurança.

12.8 Custo e tempo

Deixei esses dois requisitos não funcionais por último, pois são eles que limitam todas as outras decisões. Praticamente todas as decisões que precisamos tomar durante o desenvolvimento de software são balizadas por restrições de *custo* e de *tempo*.

Se o **custo** não fosse um fator limitante, poderíamos contratar uma equipe grande com as melhores pessoas desenvolvedoras do mundo, utilizar as melhores ferramentas de desenvolvimento, manutenção e operação de software, e usar uma infraestrutura extremamente robusta que faria com que praticamente não precisássemos nos preocupar com desempenho, escalabilidade ou confiabilidade na operação da aplicação.

A mesma coisa em relação ao tempo: se o tempo para desenvolver um software fosse ilimitado, poderíamos testar nossa aplicação por quanto tempo fosse necessário até termos certeza de que não há nenhuma falha na confiabilidade da aplicação nem vulnerabilidades que pudessem ser exploradas. Também poderíamos testar todas as possíveis arquiteturas de software conhecidas e verificar qual se adapta melhor ao problema que queremos resolver. Obviamente isso não é possível, e é devido ao tempo e ao custo de desenvolvimento de software que precisamos considerar todos os *trade-offs* entre os requisitos não funcionais de uma aplicação.

Todo projeto de software real precisa considerar sempre custo e tempo de desenvolvimento de forma realista. Não adianta prometer ter a aplicação mais eficiente e completa possível em um tempo curto e com baixo custo. Os processos de desenvolvimento como Scrum, XP e Kanban foram pensados exatamente para que seja

possível gerenciar todas essas variáveis de forma mais clara. Se nenhum processo bem definido for usado, custo e tempo normalmente são simplesmente chutes com uma data de entrega que, por sorte, pode até ser atingida, mas o mais provável é que o projeto tenha atrasos e ultrapasse o orçamento planejado inicialmente.

Os processos de desenvolvimento de software têm como um dos objetivos facilitar o acompanhamento do custo e tempo no desenvolvimento de software. Por exemplo, no Scrum, uma das principais atividades é o gerenciamento do *backlog*, que contém os itens que devem ser desenvolvidos durante o projeto. Com esse gerenciamento, é possível priorizar as tarefas que são mais importantes, aliado à **entrega contínua**, que também é um dos fundamentos do Scrum e que permite a visualização e a atualização dos custos e do tempo necessários para a finalização dos requisitos da aplicação. Dificilmente a estimativa inicial será seguida à risca, por isso é essencial utilizar metodologias como o Scrum que permitem o acompanhamento do andamento do projeto.

Utilizar metodologias para o desenvolvimento de software é essencial para manter o projeto dentro do planejado, ou para que as alterações no planejamento sejam feitas de forma estruturada. Eu não cobri esse tópico mais profundamente ao longo deste livro, pois o objetivo é entender os requisitos não funcionais que afetam a execução da aplicação. Porém, entender bem como os processos de software funcionam é também bastante importante para uma pessoa arquiteta de software, pois ela participará de diversas decisões sobre esse processo — e muitas dessas decisões da pessoa arquiteta devem ser tomadas com base no andamento do projeto.

Conclusões

Garantir que a aplicação atenda perfeitamente a todos os requisitos não funcionais é praticamente impossível. Mesmo os softwares mais complexos, que parecem ter uma escalabilidade praticamente infinita, possuem problemas. Muitas vezes, para atender a um

requisito de forma plena, precisamos diminuir um pouco a atenção em outro. Por exemplo, podemos ter um monitoramento praticamente perfeito de todas as ações que são feitas na aplicação, porém isso vai afetar o desempenho e, talvez, a escalabilidade da aplicação. É papel da pessoa arquiteta de software tomar as decisões que vão levar a aplicação a atender os requisitos não funcionais da melhor forma possível.

Este livro passou por vários dos requisitos que a pessoa arquiteta de software deve conhecer para conseguir priorizar o que é mais importante para a aplicação, para poder ajudar o restante da equipe de desenvolvimento e, também, manter o desenvolvimento da aplicação (ou de seus requisitos) dentro do tempo e do custo estimados para isso. Obviamente a pessoa arquiteta de software não trabalha sozinha: ela vai ter que conversar com vários papéis diferentes dentro de uma equipe, como desenvolvedores, gerentes, coordenadores, designers e até usuários das aplicações. Por isso, é essencial que a pessoa arquiteta de software conheça bem todos os requisitos que foram estudados aqui, para que ela consiga entender como a aplicação está sendo executada, quais os possíveis problemas que serão enfrentados e como eles podem ser resolvidos.